UNIVERSIDADE DE LISBOA
INSTITUTO SUPERIOR TÉCNICO

Data Placement in Distributed Systems

João Gonçalves Paiva

Supervisor: Doctor Luís Eduardo Teixeira Rodrigues

Thesis approved in public session to obtain the PhD Degree in
Information Systems and Computer Engineering
Jury final classification: Pass with Merit

2015

TÉCNICO LISBOA

UNIVERSIDADE DE LISBOA
INSTITUTO SUPERIOR TÉCNICO

Data Placement in Distributed Systems

João Gonçalves Paiva

Supervisor: Doctor Luís Eduardo Teixeira Rodrigues

Thesis approved in public session to obtain the PhD Degree in
Information Systems and Computer Engineering
Jury final classification: Pass with Merit

Funding Institutions

Fundação para a Ciência e Tecnologia

2015

Agradecimentos

Antes de tudo, quero agradecer ao meu orientador, o Professor Luís Rodrigues. Não tenho dúvidas que sem todo o seu apoio, motivação, e inabalável optimismo, o trabalho resumido neste documento nunca teria chegado a existir. Obrigado pelas inúmeras oportunidades. Ao longo destes anos, pude partilhar da sua sabedoria, caminhar entre os grandes e experimentar ser autónomo com a confiança de ter sempre uma sólida rede de segurança. Espero ter aproveitado da melhor forma possível.

Os meus mais sinceros agradecimentos vão também para o Professor Paolo Romano. A tua inesgotável criatividade e imparável energia foram vitais num período crítico deste trabalho, e foram em grande parte responsáveis por muitas das mais importantes contribuições desta tese. Estou certo de que a tua paixão pela investigação te levará muito longe.

Devo também um agradecimento especial ao Pedro Ruivo. A tua incrível generosidade e paciência foram vitais para que muito deste trabalho deixasse de ser apenas ideias no papel e pudesse "andar à roda" em toda a sua glória. Espero um dia ter oportunidade de te retribuir a sucessão de favores para que te disponibilizaste ao longo estes anos.

Quero também agradecer às restantes pessoas com quem colaborei: João Alveirinho, Nuno Diegues, Manuel Ferreira, João Leitão, Zhongmiao Li, e Muhammet Orazov. Trabalhar convosco ensinou-me muito e fez-me crescer.

Agradeço também ao INESC-ID, cujas óptimas condições, assim como a camaradagem dos colegas providenciaram um ambiente fértil onde este trabalho pôde crescer.

Finalmente, agradeço a todos os meus amigos e familia, que sempre me apoiaram incondicionalmente e contribuiram largamente para a manutenção da minha sanidade durante esta viagem.

Lisboa, 27 de Janeiro de 2015
João Gonçalves Paiva

To Maria, António, Fátima and Mariana,
my personal heroes

This work was partially supported by Fundação para a Ciência e Tecnologia (FCT) via the individual Doctoral grant SFRH/BD/63387/2009, via project PEPITA (PTDC/EEIS-CR/2776/2012), and via the INESC-ID multi-annual funding through the PIDDAC Program fund grant, under project PEst-OE/EEI/LA0021/2013.

Resumo

O problema da colocação de dados corresponde a decidir como atribuir dados a nós num sistema distribuído de forma a optimizar um ou mais critérios de desempenho, como a redução da congestão da rede, melhorar o balanceamento da carga, entre outros.

Ao colocar os dados perto dos clientes, podemos reduzir o número de acessos remotos, reduzir a latência das operações e evitar o congestionamento da rede. Considerando a capacidade dos nós e a caracterização dos padrões de carga, podemos evitar a sobrecarga de alguns nós que poderiam tornar-se um ponto de congestão do sistema. Se nos focarmos na probabilidade de falha de nós individuais, podemos colocar os dados de forma a maximizar a sua disponibilidade, enquanto reduzimos os custos extra da monitorização e dos múltiplos restauros de réplicas. A maioria destes critérios impõem requisitos contraditórios e cada aplicação deve dar prioridade a como optimizar a colocação dos dados. Este trabalho aborda estes e outros critérios, não só de uma forma independente, mas também combinando-os.

No entanto, os benefícios alcançados por uma boa política de colocação de dados têm de ser ponderados em função dos custos da pesquisa de dados. Na verdade, para suportar flexibilidade total na colocação de dados temos de recorrer a uma forma de diretoria distribuída, que guarda o mapeamento entre dados e nós. Infelizmente, os custos de efetuar pesquisas na diretoria e de manter a diretoria atualizada podem facilmente tornar-se num ponto de congestão do sistema. Como consequência deste problema, muitos sistemas usam estratégias de colocação de dados simples, como as funções de dispersão.

Esta tese propõe técnicas que oferecem diferentes compromissos entre esquemas simples de funções de dispersão e sistemas de diretorias completas para diferentes escalas de sistemas. O objectivo principal é fornecer melhores opções entre ter uma grande flexibilidade com escalabilidade limitada (tipicamente usado em sistemas de centros de dados) e ter uma boa escalabilidade com flexibilidade limitada (a escolha principal para sistemas à escala da internet).

Abstract

Data placement refers to the problem of deciding how to assign data items to nodes in a distributed system to optimize one or several of a number of performance criteria such as reducing network congestion, improving load balancing, among others.

By placing data near the clients, one may reduce the number of remote accesses, significantly reduce the latency of operations, and avoid network congestion. By taking into account the capacity of nodes and the workload characterization, one may avoid the overload of a few nodes that could otherwise become a bottleneck in the entire system. By minding the probability of failure of individual nodes, one can place data in a way that maximizes its availability, while reducing the overhead caused by monitoring and multiple replica restores. Most of these criteria impose conflicting requirements and each application must prioritize how to optimize placement. This work addresses these criteria among others, in an independent as well as in a combined way.

However, the benefits achieved by a clever data placement must be weighted against the costs of data lookup. In fact, to support total flexibility in the data placement, one needs to resort to some form of distributed directory that stores the mapping between data items and nodes. Unfortunately, the costs of performing directory lookups and the overhead of maintaining the directory up-to-date can easily become the bottleneck. Due to this problem, many practical systems use simple data placement strategies, such as consistent hashing.

This thesis proposes techniques that provide different tradeoffs between plain consistent hashing schemes and full directory systems for different sizes of system scales. The main goal is to provide better options between having strong flexibility with limited scalability (typically employed in datacenter systems), and having good scalability with limited flexibility (the main choice for internet scale systems).

Palavras Chave

Sistemas Distribuídos

Colocação de Dados

Armazenamento Chave-Valor

Replicação

Balanceamento de Carga

Escalabilidade

Optimização

Adaptação Autonómica

Sistemas Entre-Pares

Gestão de Dados Distribuídos

Keywords

Distributed Systems
Data Placement
Key-Value Storage
Replication
Load Balancing
Scalability
Optimization
Autonomic Adaptation
Peer-To-Peer Systems
Distributed Data Management

Index

1 Introduction **1**

1.1 Problem Statement 3

1.2 Summary of Contributions 4

1.3 Results 4

1.4 Structure of the Document 5

2 Background **7**

2.1 Basic Concepts 7

2.2 Concerns in Data Placement 8

2.3 Data Placement Dynamics 10

2.4 Optimal Placement 11

2.5 Data Location 12

2.6 Mapping Functions 13

2.6.1 Single-Valued functions 13

2.6.2 Set-Valued Functions 15

2.7 Abstracting Physical Nodes 16

2.8 Case Studies 17

2.8.1 Availability-aware Replica Placement 17

2.8.2 DataDroplets 18

2.8.3 Dynamo 18

2.8.4 Scatter 19
2.8.5 Spanner 19
2.8.6 Ursa 20
2.8.7 Case Studies Comparison 22
2.9 Summary 23

3 Internet Scale Data Placement **25**
3.1 Context 26
3.1.1 Data Replication on DHTs 27
3.1.2 Load Balancing on DHTs 28
3.1.3 Combining Structured and Unstructured Overlays 28
3.2 Rollerchain Overview and Building Blocks 29
3.3 Rollerchain Operation 30
3.3.1 Definitions and Basic Operation 30
3.3.2 The Unstructured Layer 31
3.3.3 The Structured Layer 32
3.3.4 Layer Interoperation 33
3.3.4.1 Virtual Link Creation 33
3.3.4.2 Virtual Node Maintenance 34
3.3.4.3 Virtual Node Division 35
3.3.4.4 Virtual Node Merge 35
3.3.5 Key/Value Pairs Replication 36
3.3.6 DHT routing 37
3.4 Topology Evaluation 37
3.4.1 Experimental Parameters 37

3.4.2 Fault-Tolerance Comparison 38
3.5 Replication Policies 41
3.6 Tradeoffs in Data Replication 42
3.7 A Catalog of Previous and New Policies 44
3.7.1 Policy Classification 44
3.7.2 Previous Replication Policies 46
3.7.2.1 Strategies based on consistent hashing 46
3.7.2.2 Directory-based strategies 47
3.7.3 Novel Policies 48
3.7.3.1 Oblivious Strategies 48
3.7.3.2 Informed Strategies 49
3.7.4 Summary of All Policies 50
3.8 A Performance Model To Compare Policies 50
3.8.1 System Parameters 51
3.8.2 Baseline System 51
3.8.3 Actual System 52
3.8.4 Policy Efficiency 52
3.9 Evaluation 53
3.9.1 Simulation Settings 53
3.9.2 Results 54
3.9.3 Detail 56
3.10 Summary 59

4 Datacenter Scale Data Placement **61**

4.1 Context 63

4.2 System Characterization 65

4.3 AutoPlacer Overview 66

4.3.1 Optimizer 68

4.3.1.1 Space-Saving Top-k algorithm 69

4.3.1.2 Using Approximate Information 70

4.3.1.3 Accelerating the solution of the optimization problem 71

4.3.2 Probabilistic Associative Array: Abstract Data Type Specification 72

4.3.3 The AutoPlacer iterative algorithm 73

4.3.4 Handling dynamic workloads 77

4.4 Probabilistic Associative Array Internals 79

4.4.1 Building Blocks 79

4.4.2 FeatureExtractor Key Interface 80

4.4.3 PAA Operations 82

4.4.4 Example use of PAAs in AutoPlacer 84

4.5 Optimizer Analysis 86

4.6 Evaluation 88

4.6.1 Experimental Settings 89

4.6.2 Probabilistic Associative Array 90

4.6.3 Leveraging from Locality 92

4.6.4 Distributed Optimization 94

4.6.5 Dynamic workloads 96

4.6.6 GeoGraph evaluation 98

4.7 Summary 100

5 Final Remarks **103**

5.1 Research Collaborations 104

5.2 Future Work 105

Bibliography **107**

List of Figures

2.1 Single-Valued functions. Nodes are represented by circles, data items by rectangles. 14

2.2 Set-valued functions. 16

2.3 Abstractions from physical nodes 17

3.1 Rollerchain's Architecture: At the DHT level vnodes form a ring. Each vnode is composed of several pnodes. 31

3.2 Management of virtual links among vnodes. 34

3.3 Replication signalling costs per node, considering 128bit keys. 39

3.4 Average number of nodes contacted by nodes running multi-publication, considering $R = 10$. 39

3.5 Topology signalling costs per node. 40

3.6 Load balancing vs monitoring costs tradeoff 43

3.7 Load balancing vs data transfer costs tradeoffs 43

3.8 Behaviour of policies over time 57

4.1 Example use of PAAs in AutoPlacer 84

4.2 PAA Performance. 91

4.3 AutoPlacer performance 93

4.4 Throughput of AutoPlacer, a directory-based and a consistent hashing-based solution, after a complete optimization process. 93

4.5 Progression of the quality of the data placement optimization solutions of AutoPlacer's distributed optimization versus that of a centralized optimizer, relative to unoptimized placement. 95

4.6 Total throughput of AutoPlacer over time for a dynamic workload with 90% of locality. T1 and T2 mark the instants when changes in workload occurred. . . 97

4.7 Number of objects moved to a specific node per round over time. The figure presents data for one of the nodes whose data is not affected by the first workload change (at T1). 97

4.8 Evolution of throughput over time, duration of read-only transactions, and number of remote get operations per read-only and update transactions – GeoGraph benchmark. 99

List of Tables

2.1 Systems summary table. Case studies are highlighted in **bold.** 21

3.1 Percentage of objects reachable at the end of the simulations. 38

3.2 Design Space of Replication Policies for P2P Systems 45

3.3 Policy Map 50

3.4 Evaluation Parameters 53

3.5 Results for Evaluated Policies. 54

4.1 Parameters used in the ILP formulation. 69

4.2 PAA Interface. 72

4.3 Re-located objects and size of different PAA implementations. 90

1 Introduction

In this work we consider large-scale distributed systems, composed of several nodes that do not share memory and communicate exclusively by message passing. Each node has an amount of local storage, that can be used to store a given number of data items. Collectively, the system has to store a large dataset, much larger than the amount of storage available in each individual node. Thus, different data items may have to be stored on different nodes. Furthermore, for fault-tolerance reasons, the same data items may be stored (replicated) in more than one node.

In this context, the *data placement problem* consists of selecting which nodes keep copies of each data item.

Data placement has a significant impact on several performance criteria of a distributed system, such as latency in data access, throughput, fault-tolerance, or effective storage capacity. As we will discuss, data placement decisions may affect the above properties in conflicting ways. We motivate this problem with a few simple examples:

- When a client is connected to a given node and requests data items that are stored exclusively on that node, no communication with other (remote) nodes is required to fulfill the requests. Conversely, every time the client needs to access a data item that is placed in any other system node, communication is required. Thus, by placing data near the clients, one may reduce the number of remote accesses, significantly reduce the latency of operations, and avoid network congestion. It is then clear that data placement may have impact on the performance of the applications that execute on the distributed system.
- If data is replicated, replicas need to be accessed in a way that they appear to be mutually consistent. This requires non-trivial coordination among the nodes that store replicas of a data item during write operations, read operations, or both. Thus the performance of the system depends on the number of replicas, the replication protocol, and also on the ratio between read and write operations. In any case, replication usually involves a

tradeoff between the efficiency of read operations (that in some cases can be local and uncoordinated) and the cost of update operations (that require to update multiple nodes).

- In a large dynamic system, failures are unavoidable. To prevent data from being lost, the system must react to failures by ensuring that all data items have an appropriate number of replicas at all time. Yet, as shown by Blake and Rodrigues (Blake and Rodrigues, 2003), this can be costly in terms of bandwidth. On the other hand, it is possible to predict failures. By minding the probability of failure of individual nodes, one may place data in the more resilient ones to maximize data availability, while reducing the overhead caused by monitoring and multiple replica restores. In fact, this presents a tradeoff since it may cause the more resilient nodes to become a bottleneck in the system, while other nodes more prone to failure may remain unused.

Thus, choosing the right data placement strategy usually involves making tradeoffs among different performance criteria. Most of these criteria impose conflicting requirements and each application must prioritize how to optimize placement. This work addresses the aforementioned criteria, among others, in an independent and in a combined way.

Furthermore, when considering the data placement problem, one also has to consider the costs of the mechanisms that collect the information required to make informed placement decisions and the mechanisms required to keep track of the location of data (after placement has been decided). This is illustrated by the following considerations:

- If data placement does not take into consideration how data items are accessed by the applications, this may result in an uneven distribution of load among the nodes, because some data items may be accessed much more frequently than others. On the other hand, if access patterns are taken into account, one may be forced to collect and distribute monitoring information to identify those patterns, a task that may consume a non-negligible amount of existing resources.

- As it will be discussed, a problem that is tightly related to the data placement is the data location problem. This problem consists of finding which nodes store a given data item, once the placement has been defined. The most obvious way of solving the data location problem consists of using a centralized directory, which keeps the mapping among data items and nodes. Unfortunately, such central directory can easily become a bottleneck in the system.

The benefits achieved by clever data placement must be weighted against the costs of data lookup. In fact, to support total flexibility in the data placement, one needs to resort to some form of distributed directory, that stores the mapping between data items and nodes. Unfortunately, the costs of performing lookups to the directory and the overhead of maintaining the directory up-to-date can easily become the bottleneck. Due to this problem, many practical systems use simple data placement strategies, such as consistent hashing (Karger et al., 1997).

Following the above observations, in this thesis, we identify the key issues in data placement and make an overview on how these issues have been previously addressed in the literature. We then propose new techniques that provide different tradeoffs between plain consistent hashing schemes and full directory systems for different sizes of system scales. We studied two scenarios of system scales: the Internet scale and the Datacenter scale. For each scale, we propose techniques that improve over the tradeoffs provided by the state of the art, by providing greater flexibility at cost of no scalability for internet-scale systems and by providing greater scalability with good placement flexibility for datacenter-scale systems. One might think that there would be a unifying data placement strategy for both scenarios. However, our experience appears to indicate that such a goal cannot be achieved. We discuss these issues in Section 5.

1.1 Problem Statement

The problem of optimizing data placement in a distributed system is very complex. It involves not only making tradeoffs with regard to the properties directly influenced by the location of the data (latency, throughput, among others), but also tradeoffs with regard to the amount of information that needs to be gathered and maintained to make informed placement decisions and keep track of item locations. All these factors may impose limits on the scalability of a given solution, whether in terms of the number of nodes that can be supported, or the amount of data location data managed, or the granularity at which data placement decisions take place.

The main goal of this thesis is to explore the tradeoffs between scalability and full placement flexibility, by proposing novel designs to balance these conflicting goals.

1.2 Summary of Contributions

The contributions of this thesis are two techniques of data placement for internet-scale and datacenter-scale systems:

- For internet-scale systems, the thesis proposes a network overlay composed of groups of nodes. This overlay provides enough flexibility to support pluggable policies that optimize a range of performance criteria including improving load balancing, bandwidth usage and monitoring costs. Furthermore, this increased flexibility does not come at the cost of limiting scalability and, in fact, our solution is more reliable under heavy network dynamics than existing techniques.
- For datacenter-scale systems, this thesis proposes a hybrid approach which combines consistent hashing for most items with an efficient directory for hotspot items. This design is more scalable than using a directory to define the placement of all items, but still provides enough flexibility to exercise fine-grained control over data placement. This fine-grained control was used to implement a distributed algorithm improving data access locality on a key-value store.

1.3 Results

Considering the contributions listed above, the main results of this thesis are the following:

- Experimental prototype of a virtual-node distributed key-value store which addresses concerns of node replacement, scalability and load balancing (dubbed Rollerchain).
- Design and implementation of a set of replication policies for internet-scale systems that leverage the virtual-node design of Rollerchain.
- Extensive experimental evaluation of a prototype of Rollerchain as well as other related state-of-the-art systems through simulation.
- Extensive experimental evaluation of a set of replication policies for virtual node-based systems through simulation.

- Experimental prototype of a distributed key-value store which addresses concerns of scalability and data locality (dubbed AutoPlacer).
- Extensive experimental evaluation of AutoPlacer using real-word deployments on a dedicated cluster.

1.4 Structure of the Document

The document is structured as follows. In Chapter 2, we identify the main concerns that need to be taken into account when performing data placement and list the most relevant placement strategies that have been proposed in the literature. In Chapter 3, we present our contributions for internet-scale systems. Chapter 4 describes our solution for datacenter-scale systems. Finally, we conclude the document and present lines of future work with Chapter 5.

2 Background

This chapter provides an overview of the main concepts, concerns and systems that served as inspiration for the work presented in this thesis. We first introduce some basic notions in data placement, then we address some key concerns and present the dynamics of data placement. Next, we introduce the related data location problem and we discuss the topic of optimal placement. Finally, we present the main techniques used to map data items to system nodes and conclude by reviewing the most relevant related systems.

2.1 Basic Concepts

In this section, we introduce a number of concepts that are relevant to discuss the data placement problem.

Data item We denote the atom of data storage a data item. In a concrete system, an item may be materialized as a data object at a different level of abstraction: a block, a file, a programming language object, a portion of a relational table, among others. In this thesis, we abstract from such materialization. We assume that each data item has some unique identifier that we will further refer to as the item's unique *key*.

Data owner: We denote the node where a given data item i is placed as the i's *data owner*. Therefore, data placement can be informally described as the problem of deciding which nodes own which data items.

Replication: When the same data item is stored in multiple nodes, we say that the data item is replicated. The number of replicas stored of a given data item is named the item's *replication degree*. As we will discuss later in the text, in principle, nothing prevents different items from the data set to be stored with different replication degrees. Still, in many practical systems, the replication degree is a global constant that is applied to all data items. The particular case where all data items are replicated in all nodes is named *full replication.*

Types of Data Access: We distinguish two main types of operation on data items: *Read* operations, which do not change the value of the data item; and *Write* operations, also named *updates*, which change the value of the item.

Replica Consistency: A *replica consistency model* defines which outcomes are legal for any sequence of read and write operations. Ideally, a replicated system would behave like a centralized system, i.e., it would provide *one-copy equivalence*, as this is an intuitive memory model for the programmer. A *data consistency protocol* is a protocol to be executed during read and write operations to enforce a given data consistency model. To simplify the exposition, in this document, we assume that when items are replicated, a *read-one write-all* protocol is used to ensure replica consistency. However, most of the discussion included in this thesis is still valid if other consistency models and/or consistency protocols are used.

Cache: A cache copy of a data item is a transient replica whose existence is not required to ensure data survivability and that can be created and discarded with minimal coordination. We do not discuss caching techniques; they can be seen as complementary to the replication and placement techniques studied in this thesis and we direct the interested reader to Liu and Maguire, 1994.

2.2 Concerns in Data Placement

This section enumerates the key concerns that need to be addressed when performing data placement.

Scalability: Any data placement scheme that does not rely on full replication has the potential to increase the scalability of the system. In fact, since each node only stores a portion of the data set, as more nodes are added to the system, the size of the data set that can be stored in the entire system can also grow. Exclusively from the point of view of scalability, the smaller the replication degree of an item, the better the scalability of the system, since less space is occupied by each item.

Load Balancing: By placing different data items in different nodes, it is possible to have programs that access those data items to execute in parallel, thus distributing the load among multiple servers.

Furthermore, replication also allows to distribute the load of read accesses to a single data item by multiple replicas. This is typically an advantage, since most workloads are read-intensive. However, write accesses require all replicas to be updated, and therefore do not benefit from replication (actually, the cost of write operations may increase with the number of replicas). Therefore, there is a tradeoff between the gains that can be achieved during read operations and the overhead imposed during writes.

Ideally, instead of simply distributing the load among the available servers, one would like to *balance* the load, i.e., to ensure an even load distribution among the servers. Given that, in most cases, some data items are more accessed than others, load balancing requires the relative rate of access to different items to be estimated and taken into account. Furthermore, the access patterns may not be constant in time, which may require the data placement to be recomputed from time to time.

Data Locality: As noted before, a program that executes at a given node may be required to exchange messages with other nodes when accessing data items that are not available locally. Typically, the smaller the number of nodes that need to be contacted for executing a program, the better. Therefore, data placement should strive to put data items that are often accessed together in the same node(s).

It is interesting to note that data locality may conflict with the load balancing concern discussed above. In fact, if the data items that are accessed together by some programs are also the data items that are accessed most frequently in the system, putting those items in a small set of nodes may overload these nodes. However, it might still be beneficial to place said data items in nodes closest to the ones running the programs that use them, in terms of some network metric (e.g. latency).

Availability: There are two main reasons to use data replication in a system. One is to distribute the load of read operations, as discussed before. The other is to ensure data availability in case of node failures. In fact, if data is replicated in multiple nodes, data is not lost when a node crashes.

If the probability of failure is uniform, and failures are not correlated, data placement has only to ensure that replicas of a given item are placed in different nodes. Otherwise, failure probabilities and failure correlation should be taken into account. For instance, replicas should be placed in nodes whose failure correlation is small (for example, in different racks). Also, items that are stored in highly reliable nodes may have a smaller replication degree than items that are stored in less reliable nodes.

Again, note that availability concerns may conflict with other goals. For instance, a higher replication degree may increase data availability but may also increase the cost of write operations. Also, if data placement is skewed to more reliable nodes, these nodes can easily become overloaded.

Node Replacement: When a node fails, the replicas that it owned must be reconstructed, either at other surviving nodes, or in a new node that is brought to life to replace the failed one. The recovery procedure involves initiating the new replicas by copying their state from other surviving replicas. This operation can take some time and consume a non-negligible amount of system resources. Naturally, recovery time is a function of the amount of data replicas that were stored on the failed node (and that have not been lost) and must be re-replicated. As a result, a more predictable recovery time can be achieved if the number of data items is distributed evenly among the existing nodes.

Node Population: When a node joins the system, it must be populated with replicas of data items that should be copied from other nodes. The strategy used for data placement can also affect the amount of time needed to populate the new node. For instance, in systems that exhibit an asymmetry between upload and download bandwidths, the population time can be shortened if the new node can download different items in parallel from different sources (this discourages a placement strategy that makes a new node a mirror of a single existing node). Also, if network latencies are heterogeneous, the population time can be shortened if the items placed in the new node come from low-latency neighbors.

2.3 Data Placement Dynamics

Another concern is how data placement must change over time. We consider two types of mappings: static and dynamic.

Static In a system using static data placement, the location of data items is defined a priori and maintained through the lifetime of the system. If a node crashes, the system waits until the node recovers or is replaced by a new node that will contain exactly the same replicas as the failed one.

Dynamic In a system using dynamic data placement, data placement is redefined from time to time. The three main causes for triggering a new data-placement are changes in:

- **System membership:** when nodes are removed or new nodes are added, for example as part of elastic scaling. When the scale of the system changes, not only may data items be replaced, but also the replication degree of items may be changed;
- **Underlying network:** When the network condition changes, data may be moved to minimize the costs of updates;
- **Workload:** On workload-driven systems, a new data placement may be defined in response to changes in the node's data access patterns that may affect locality of data or the most accessed items.

2.4 Optimal Placement

As already hinted in the paragraphs above, several of the listed concerns point toward conflicting goals. Thus, in a concrete system, the administrator must define the tradeoffs that are more relevant for the system operation, considering the concrete requirements of the applications that are being executed.

Optimal placement could then be defined as a cost function, that would assign a weight to each concern and a numeric value to each configuration for every relevant concern. Optimal placement would be the one that would minimize the cost function. In general, this an NP-hard problem (You et al., 2013).

It is worth noting that most systems do not attempt to achieve optimal placement, because it can be too expensive and slow to derive or simply because, for that system, it may be difficult to express a meaningful cost function. For instance, if the data location cost dominates the cost function, the placement may be simply guided by the choice of an effective mapping function (such as consistent hashing), discarding all other factors.

2.5 Data Location

After the data has been placed, it is necessary to keep a record of the location of the replicas of each data item, such that these replicas can be located when the data needs to be read or updated.

Any arbitrary placement can be registered in a *directory* service. Logically, the directory service keeps a table, indexed by the data item's key, that stores, for each item, the number of replicas and the nodes where these replicas are placed. A directory server can be implemented in several ways, ranging from a single centralized node to a partially replicated distributed hash table hosted on thousands of nodes. This solution allows for perfect placement for a scenario where the system stores few items items on nodes with stable connections.

As the system grows in items, the directory service can easily become a bottleneck for system operation. If we consider a system with a large number of data items, the directory can easily become too large to be stored on a single node. Hence, if it is replicated at every node, it may consume a significant amount of memory and the costs of updating an entry, particularly when the placement is dynamic, may become prohibitive. On the other hand, if the directory is maintained on dedicated nodes, as a distributed or centralized service, an additional roundtrip is required to locate items, increasing network congestion and introducing latency in the critical path to accessing data.

If we consider a system with a large and unstable membership, the directory must be updated each time a node joins or leaves the system. This problem is complicated when one distributes the directory by the nodes storing the data as then, not only the data, but also the directory, must be replicated and kept consistent despite frequent node joins and leaves.

Since the solutions studied in this thesis are focused on internet-scale systems characterized by highly dynamic memberships, as well as datacenter systems which store a large number of items, using directories to define full data placement may be an unsuitable solution. Nevertheless, in the following chapters, we will also address more in depth how our techniques compare with directory-based techniques.

2.6 Mapping Functions

As discussed in the previous section, any arbitrary mapping among data items and nodes can be captured by a directory. Unfortunately, directories are expensive to implement. Therefore, many systems rely on alternative *mapping functions*, that can be implemented in a very efficient manner, such that data location can be performed with minimal overhead, even if the resulting data placement is far from optimal.

We define data placement as the graph D in the relation (K, N, D) between the set of all keys, K, and the set of nodes in the system, N. This relation is left-total, i.e., all keys have at least one corresponding node in the system; as well as symmetric, i.e., if a key k corresponds to a node n in the graph D, then the node n also corresponds to the key k in the graph D.

One can devise two mechanisms for defining the data placement graph using mapping functions: (I) mapping each key to a node using a single-valued function; (II) mapping each key to a sub-set of N by using set-valued functions;

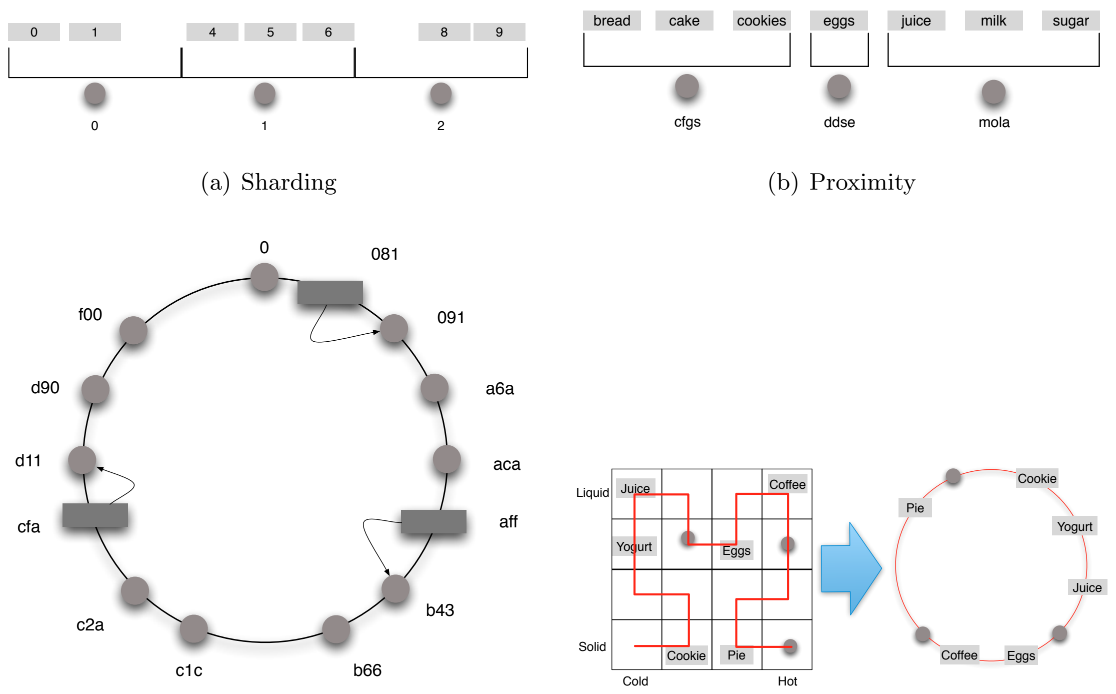
2.6.1 Single-Valued functions

Single-Valued functions are deterministic functions that typically take as argument the key's identifier and/or attributes and return the node where the data item should be placed. Note that, since a single value is returned, when a data item is replicated, each replica needs to have its own unique identifier that can be fed to the mapping function to derive the location of that replica.

In this section, we will study some of the most representative examples of these functions.

Sharding: Sharding (Agrawal et al., 2004; Curino et al., 2010) is the simplest technique to perform data placement. All data replicas are sorted using one of the attributes (such as the key) or a combination of attributes (such as the owner and the key). Then the ordered set is divided into $|N|$ chunks using cut points in the ordered set and each chunk is placed on a different node (to simplify location, nodes are also typically sorted by their identifiers and the first chunk goes to the first node, etc) (Figure 2.1 (a)).

Proximity Placement: As before, proximity placement (Harvey et al., 2003; Schutt et al., 2006; Lakshman and Malik, 2010) starts by ordering all replicas using a combination of



(a) Sharding (b) Proximity

(c) Hashing (d) Space Filling Curves

Figure 2.1: Single-Valued functions. Nodes are represented by circles, data items by rectangles.

attributes. The combination of attributes defines a placement identifier for each data item. Furthermore, each node is given an identifier selected from the same identifier space of the data items. Finally, instead of splitting the ordered set of items using cut points, data items are placed in the nodes that have the closest identifier to that of the item (Figure 2.1 (b)).

One problem with this approach is that, if the identifiers of the nodes are taken uniformly at random from the identifier space but the distribution of the data item identifiers is not uniform, this results in a very imbalanced item distribution. Thus, the placement can and should be controlled by carefully selecting the node identifiers (Kenthapadi and Manku, 2005).

Consistent Hashing: Consistent hashing (Karger et al., 1997) consists of generating a data item's key by hashing its contents (using a cryptographic hash function) and a node's identifier by hashing one of its unique characteristics (e.g. its IP address) so that data items can be placed deterministically in those nodes whose identifier is closest to the item's key (Figure 2.1 (c)). This way, when the membership changes, only nodes with identifiers closest to the identifiers of the removed or added nodes are affected, since they become responsible for more or less data, respectively.

Note that consistent hashing is very similar to proximity placement, except that the hashing function can help in masking an uneven distribution of data identifiers in the identifier space.

Space-Filling Curves: In order to support operations requiring ordering over several attributes (e.g. the intersection of two requests which request data items whose attribute values fall in specific ranges for two different attributes), some systems make use of Space-Filling Curves (Moon et al., 2001). This technique maps a multidimensional identifier space to a unidimensional one, where data items and nodes are placed similarly to what happens in systems using order-preserving partitioning (Figure 2.1 (d)).

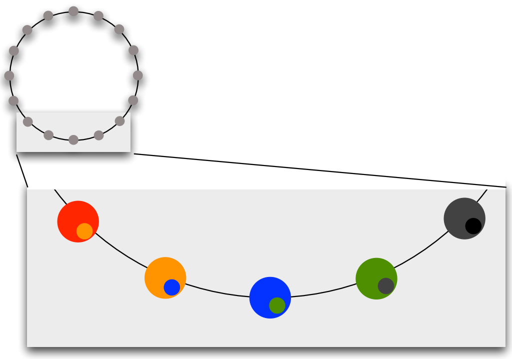
2.6.2 Set-Valued Functions

Set-valued functions are deterministic functions that, as before, take as argument the key's identifier and/or attributes and return a set of nodes where the data item should be placed. The advantage of using set-valued functions is that replicas of a given item do not need to be explicitly named: their location is provided directly by the mapping function. In this section, we will study some of the most representative examples of these functions.

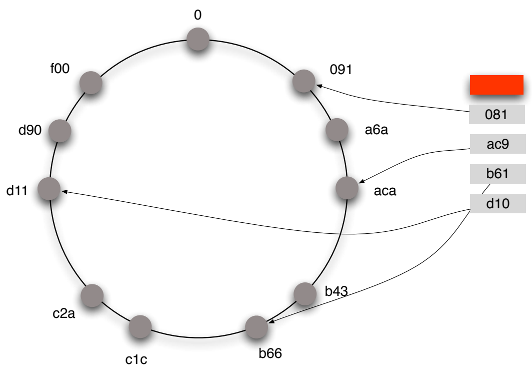
Neighbor Replication: Neighbor replication (also known as leaf set replication) assumes that the nodes are organized in some overlay topology, such that it is possible to easily name a set of k neighbors of a given node. For instance, if nodes are organized in a Chord ring (Stoica et al., 2001), each node always stores the data from its k successors (Figure 2.2 (a)).

Under this assumption, neighbor replication consists of using one of the single-valued functions described before, and returning that value plus $r - 1$ neighbors.

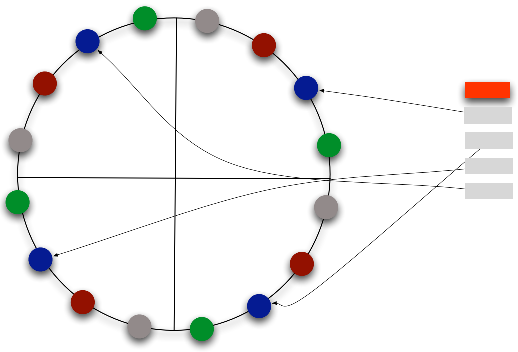
Multi-Publication: Multi-publication (Knežević et al., 2009; Ratnasamy et al., 2001; Waldvogel et al., 2003) consists of deterministically deriving r different identifiers from the input and then using one of the single-valued functions described before for each of these identifiers (Figure 2.2 (b)). Typically, this is implemented by applying consistent hashing using



(a) Neighbor Replication



(b) Multi-Publication



(c) Symmetric Replication

Figure 2.2: Set-valued functions.

r different hash functions, or the same hash function with r different salt values appended to the key.

Symmetric Replication: Symmetric replication (Ghodsi et al., 2007a; Schütt et al., 2008) is a variant of multi-publication where the identifier space is divided into r slices, and the replicas of each data item are placed one in each slice at *slice size* distance from the replica in the previous slice (Figure 2.2 (c)). This causes each node's data items to be replicated in contiguous portions of identifier space, which increases the probability that they will be stored by few different nodes.

2.7 Abstracting Physical Nodes

In the text above, we have assumed that a mapping function maps a combination of keys and attributes onto a physical node in the system. However, the same functions can be used to map onto logical nodes, which in turn, are mapped onto physical nodes by some additional mechanism. There are two main ways of implementing logical nodes: virtual nodes and virtual servers, as described below.

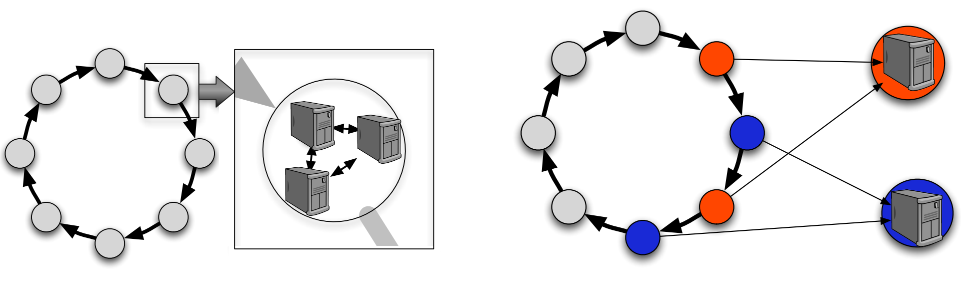

(a) Virtual Nodes (b) Virtual Servers

Figure 2.3: Abstractions from physical nodes

Virtual Nodes: In architectures based on virtual nodes, each logical node is implemented by a group of nodes (a virtual node) which acts as a single node in the system. In these architectures, all nodes in each virtual node replicate the same data (Figure 2.3 (a)).

Virtual Servers: In architectures based on virtual servers, each physical node supports multiple logical nodes (Figure 2.3 (b)). This allows to artificially augment the number of logical targets for the mapping functions and obtain a better statistical distribution of items among the physical nodes.

2.8 Case Studies

There is an extensive list of distributed replicated systems that make use of data placement. Table 2.1 presents a summary of a number selected systems which employ the previously described techniques. In the next sections, we briefly describe a representative subset of these systems illustrating different approaches to the problem. The selected systems are presented in alphabetical order.

2.8.1 Availability-aware Replica Placement

The work by Mickens and Noble, 2006 proposes techniques for exploiting node availability information on distributed systems. This work describes mechanisms to predict the availability of nodes, and how such information can benefit several applications, including reducing object copying operations in a replicated distributed system.

This work proposes placing more data in the nodes that are predicted to be most available, such that, when nodes fail, the amount of data that must be re-replicated is reduced. Since data placement cannot be defined using a deterministic function without global knowledge, this work uses a distributed directory to map keys to nodes.

As shown in the work by Pace et al., 2011, such a mechanism leads to poor load balancing, since nodes with higher predicted availability will own most of the data. To avoid this problem, the latter work also makes use of a distributed directory to achieve a similar goal. However, data items are placed in nodes with regular availability histories. RelaxDHT's (Legtchenko et al., 2009) design also aims at reducing object copying through the use of a distributed directory. RelaxDHT achieves better load balancing than the former alternatives, since data is first placed in the system through consistent hashing, and the distributed directory is used only to avoid re-replicating data after membership changes.

2.8.2 DataDroplets

The work by Vilaça et al., 2011 consists of a distributed key-value store which incorporates a locality-aware strategy for data placement. This system was designed mostly for applications where data is arbitrarily related and requested through tags (such as Twitter).

The main technique behind DataDroplets is the use of a space-filling curves mapping function, which uses tags as dimensions in the multi-dimensional identifier space. This allows objects with similar tags to be placed in logically close nodes, improving the locality of requests using tags, especially when using multiple tags at once, since the system can determine exactly which nodes are relevant for the request. Other systems, such as Andrzejak and Xu, 2002, Schmidt and Parashar, 2004, also make use of the same technique to improve the performance of range queries.

2.8.3 Dynamo

Dynamo (DeCandia et al., 2007) is a highly available key-value store developed at Amazon. Its main purpose is to store data for services which require high performance and only need the simple interface of key-value stores (as opposed to requiring complex operations typically found in SQL systems).

In Dynamo's design, scalability and availability are paramount. The data is partitioned using consistent hashing, and replicated through neighborhood replication. Consistent hashing provides the system with incremental scalability, while neighborhood replication improves availability since the node owning the data after a node's failure will already be replicating all the failed node's data. Additionally, Dynamo also employs virtual servers to improve the load balancing of the system.

Pastry (Rowstron and Druschel, 2001b), Chord (Stoica et al., 2001) and CFS (Dabek et al., 2001) are examples of other systems also making use of consistent hashing combined with neighborhood replication. Pastry also uses a small directory at each node to provide low-latency neighbors to improve the latency when requesting data items from the system.

2.8.4 Scatter

Scatter (Glendenning et al., 2011) is a highly consistent key-value store designed to be used in internet-scale deployments. Scatter provides linearizable storage, through the use of Paxos (Lamport, 1998).

This system is based on the virtual nodes technique: groups of nodes of variable size are maintained by Paxos, and data items are attributed to them through consistent hashing. Virtual nodes allow Scatter to vary the replication degree in response to changes in the membership without altering the topology of the network. To improve load balancing, scatter also allows the virtual nodes to change their identifiers to others closer or more distant from their neighbors, so that consistent hashing will cause data to be moved to their neighbors.

Elastic Replication (Abu-Libdeh et al., 2011) also makes use of virtual nodes maintained by a strong consistency protocol (in this case, chain replication (van Renesse and Schneider, 2004)) to achieve higher availability.

2.8.5 Spanner

Spanner (Corbett et al., 2012) is a scalable replicated database published in 2012 by Google. Its main goals are to provide strong consistency in datacenter-scale deployments, by leveraging on an API that exposes clock uncertainty.

Spanner partitions data items into buckets (defined as intervals of the consistent hashing key space), which are mapped to nodes through a distributed directory. This allows the system to change the location of data items to improve load balancing, geographic distribution (to reduce global placement time), or to improve data locality. PNUTS (Cooper et al., 2008), a system with similar goals, also has a similar design, which allows it to also support ordered placement of data by partitioning the ordered key space into intervals, each of which is mapped to a bucket which is then mapped by the directory to nodes.

2.8.6 Ursa

Ursa (You et al., 2013) is a scalable data management middleware designed to perform dynamic load reconfiguration of data storage systems. Its main goal is to determine the data placement that achieves best load balancing, considering the observed data access patterns, while minimizing the reconfiguration costs.

The main technique involved in this system is the use of a distributed directory to express arbitrary key-node relations of dynamic nature. The system operates by calculating the best data placement configuration using an optimization algorithm in a centralized location and then pushing the resulting “move” operations to the nodes and updating the directory. All requests are routed by first querying the directory. Should the requested objects be under the process of being relocated, the requests are delayed until the operation is complete.

Other works designed to improve data locality and global placement time, such as (Leff et al., 1993; Li and Hudak, 1989; Carter et al., 1991; Lim et al., 1997; Herlihy and Sun, 2005; Demmer and Herlihy, 1998), use similar techniques as Ursa, differing mainly on the optimization algorithm selection and implementation, which may also be of a distributed nature.

Table 2.1: Systems summary table. Case studies are highlighted in **bold.**

System Name	Reference	Scalability	Load Balancing	Data Locality	Availability	Node Replacement	Populating a new node	Dynamics	Techniques
RAID	(Patterson et al., 1988)				X			Static	Sharding
Chord	(Stoica et al., 2001)	X	X		X			Dynamic: System Membership	Hashing + Neighbor
Pastry	(Rowstron and Druschel, 2001b)	X	X		X		X	Dynamic: System Membership	Hashing + Neighbor + Directory
Dynamo	**(DeCandia et al., 2007)**	X	X		X			Dynamic: System Membership	Hashing + Neighbor + Virtual Servers
CFS	(Dabek et al., 2001)	X	X		X			Dynamic: System Membership	Hashing + Neighbor + Virtual Servers
Skipnet	(Harvey et al., 2003)	X	X	X				Dynamic: System Membership	Proximity
Chord#	(Schutt et al., 2006)	X	X	X				Dynamic: System Membership	Proximity
Cassandra	(Lakshman and Malik, 2010)	X	X	X				Dynamic: System Membership	Proximity
Squid	(Schmidt and Parashar, 2004)			X				Dynamic: System Membership	Space Filling Curves
	(Andrzejak and Xu, 2002)			X				Dynamic: System Membership	Space Filling Curves
DataDroplets	**(Vilaça et al., 2011)**			X				Dynamic: System Membership	Space Filling Curves
Scalaris	(Schütt et al., 2008)	X	X	X	X			Dynamic: System Membership	Proximity + Symmetric
	(Ghodsi et al., 2007a)	X	X		X			Dynamic: System Membership	Hashing + Symmetric
CAN	(Ratnasamy et al., 2001)	X	X		X		X	Dynamic: System Membership	Hashing + Multi Publication
	(Waldvogel et al., 2003)	X	X		X		X	Dynamic: System Membership	Hashing + Multi Publication
	(Knežević et al., 2009)	X	X		X		X	Dynamic: System Membership	Hashing + Multi Publication
Scatter	**(Glendenning et al., 2011)**	X	X		X	X		Dynamic: System Membership	Hashing + Virtual Nodes
	(Abu-Libdeh et al., 2011)	X	X		X	X		Dynamic: System Membership	Hashing + Virtual Nodes
RelaxDHT	(Legtchenko et al., 2009)				X	X		Dynamic: System Membership	Hashing + Dist. Directory
	(Pace et al., 2011)				X	X		Dynamic: System Membership	Dist. Directory
	(Mickens and Noble, 2006)				X	X		Dynamic: System Membership	Dist. Directory
Spanner	**(Corbett et al., 2012)**		X	X	X			Dynamic: Workload	Hashing + Directory
PNUTS	(Cooper et al., 2008)		X	X	X			Dynamic: Workload	Hashing + Directory + Ordered
	(Leff et al., 1993)			X				Dynamic: Workload	Dist. Directory
Ursa	**(You et al., 2013)**			X				Dynamic: Workload	Dist. Directory
Ivy	(Li and Hudak, 1989)			X				Dynamic: Workload	Dist. Directory
Munin	(Carter et al., 1991)			X				Dynamic: Workload	Dist. Directory
Arrow	(Demmer and Herlihy, 1998)			X			X	Dynamic: Workload	Hierarchical Directory
	(Herlihy and Sun, 2005)			X			X	Dynamic: Network	Hierarchical Directory

2.8.7 Case Studies Comparison

Table 2.1 captures the key properties of the systems described above and of other similar systems that we have opted to not describe in detail.

The solutions can be broadly classified into two categories, namely those making use of directories and those making use of mapping functions. Systems making use of directories typically target concerns which require more dynamic data placement, in particular those of improving data locality. This fact is due to such designs allowing for better data placement flexibility when compared with mapping functions. The exceptions to this are Pace et al., 2011, Mickens and Noble, 2006, in which directories are used to avoid data movement as a reaction to membership changes. This scenario requires more data placement flexibility than typical mapping functions, which would cause data reshuffle after membership changes.

Regarding the systems making use of mapping functions, we can identify three main categories, namely:

- systems that use Consistent Hashing (*Hashing* on the table) to achieve scalability with good load balancing, typically combined with neighbor, multi-publication or symmetric replication for availability. This design is most common in internet-scale systems, such as Stoica et al., 2001, Rowstron and Druschel, 2001b, DeCandia et al., 2007, Ratnasamy et al., 2001, mainly due to the advantages of low-cost incremental scalability of consistent hashing.

- systems that make use of proximity placement and space filling curves with the objective of clustering related data in sets of nodes to improve data locality, such as Harvey et al., 2003, Schutt et al., 2006, Lakshman and Malik, 2010, Schmidt and Parashar, 2004, Andrzejak and Xu, 2002, Vilaça et al., 2011, Schütt et al., 2008. These systems are characterized by better flexibility than those based on Consistent Hashing, and significantly better scalability and running cost than those based on directories. However, their placement flexibility is considerably lower than that of the directory-based systems similar to Ursa (You et al., 2013), which can control data placement with data-item granularity.

- systems that employ virtual nodes (Glendenning et al., 2011; Abu-Libdeh et al., 2011) to improve availability. Such architectures allow for a flexible replication degree, providing advantages for this concern, since the system may delay reactions to membership changes, leading to more stable systems.

In fact, most systems target more than one concern by combining several techniques. While the most common combination is to use some replication mechanism along with consistent hashing or proximity placement, some such as Cooper et al., 2008, Leff et al., 1993 combine consistent hashing with coarse-grained directories to obtain incremental scalability with some degree of control over data locality and load balancing.

2.9 Summary

This chapter provided an overview of the main systems that served as inspiration for the work presented in the thesis. It discussed the main techniques used to map data items to system nodes and overviewed the most relevant systems that make use of them.

3 Internet Scale Data Placement

The internet scale scenario is characterized as having millions of nodes with unstable connections on an asymmetric, unpredictable and uncontrolled network. At this scale, building distributed directories to support fine-grain placement or running optimization procedures that require global knowledge is completely unfeasible. The most prevalent way of storing data at this scale is to use a Distributed Hash Table.

Distributed Hash Tables (DHTs) (Stoica et al., 2001; Rowstron and Druschel, 2001b; Maymounkov and Mazières, 2002; Gupta et al., 2003) are *structured overlays*, i.e., overlays in which nodes organize themselves into a predefined topology that supports routing. As a result, DHTs can efficiently map a key to a peer active in the system (named the key owner). This functionality allows to implement store/lookup operations of key/value data pairs in a distributed and scalable manner. On top of this basic functionality, it is possible to build several types of internet-scale distributed applications and services, such as resource-location (Alveirinho et al., 2010), publish/subscribe (Castro et al., 2006), multicast (Zhuang et al., 2001), distributed storage (Dabek et al., 2001), among others (Ratnasamy et al., 2001; Rowstron and Druschel, 2001a). As in any other system, in DHTs nodes can voluntarily join, leave, or simply fail. Furthermore, in open internet-scale systems, these membership changes can happen at a fast pace, a phenomenon known as *churn* (Wu et al., 2006). In order to scale efficiently when facing constant membership changes, most DHTs use some form of consistent hashing (Karger et al., 1997).

Consistent hashing allows changes in membership of DHTs to be handled only by the neighbors of the joining/failed node, reducing their impact on the overlay. However, the main drawback of consistent hashing is that it does not provide any kind of placement flexibility. Nodes must join the network using identifiers created according to strict rules (typically using a derivation of their IP and port), and data items are mapped to the nodes with identifiers closest to their own. This fact effectively limits the actions that a system may take to react to changes in load or to improve other criteria. This lack of flexibility motivates the need for internet-scale solutions with more flexible placement.

In this chapter we present Rollerchain, an overlay network designed to optimize data placement at the Internet scale. Rollerchain uses a completely different strategy to improve the flexibility of data placement for internet-scale systems: it relies on consistent hashing, using a Distributed Hash Table (DHT), but makes each *logical node* of the DHT to be materialized by a set of physical nodes. The main advantage of this approach is that it supports pluggable replication policies, as several of its components can be configured to improve different performance criteria such as monitoring costs, load balancing or data transfer costs. Furthermore, these flexibility gains are not achieved by sacrificing or trading off fault-tolerance. On the contrary, experimental results show that our architecture maintains the data items reachable even under heavy churn, while previous approaches fail to preserve data availability in such conditions.

Rollerchain's placement flexibility enables the design of different replication policies that improve several distinct performance criteria. These policies can be selected depending on the application and/or environment where the system is deployed. Consequently, our design can address a larger range of challenges than other state-of-the art solutions.

The rest of this chapter is organized as follows. Section 3.1 puts the related work into the context of Rollerchain. Section 3.2 provides a brief overview on the building blocks of our solution. Section 3.3 describes the operation of Rollerchain. Section 3.4 contains results on how the overlay compares with other state of the art solutions. Section 3.5 presents the main metrics that can be improved by policies which exploit Rollerchain's flexible topology. Section 3.6 describes the challenges involved in the design of such policies. Section 3.7 overviews existing, as well as newly proposed replication policies. Section 3.8 presents a model for evaluating those policies according to multiple criteria. Section 3.9 presents the comparison of policies according to this model. Finally, Section 3.10 concludes the chapter.

3.1 Context

This section puts the related work into context, regarding Rollerchain. It focuses on how the related work handles data replication and load balancing, and reviews other works which mix unstructured with structured overlay networks.

3.1.1 Data Replication on DHTs

The most common mechanism for replicating data in DHTs is Neighbor Replication. We recall that this mechanism keeps copies of each key in the r neighbors of the key's owner. A significant advantage of Neighbor Replication is that, when its neighbors change, the key owner may trigger the creation of new replicas. Since each node keeps a different set of replicas, bookkeeping becomes costly if one attempts to use a flexible scheme where the number of replicas fluctuates. Therefore, most schemes use some variant of eager replication, where replicas have to be created when nodes, fail and moved when nodes join. This becomes very expensive in terms of bandwidth as demonstrated in Blake and Rodrigues, 2003. Furthermore, depending on the routing scheme used in the DHT, Neighbor Replication may not perform a fair load distribution, as some replicas are more likely to be hit by queries.

Multi-Publication stores r replicas of each data item in different positions of the DHT. This means that, in order to keep the data from being lost, some mutual monitoring scheme needs to be implemented to detect the departure/failure of a node containing a replica, and subsequently restore the replication degree. The main advantage of Multi-Publication is that it offers very good load balancing properties, as multiple queries may be diverted to different regions of the DHT. On the other hand, monitoring becomes expensive, because it needs to use DHT routing and a node may be forced to monitor a different set of nodes for each object that it stores.

Finally, a number of recently proposed overlays take a different approach to replication (Lynch et al., 2002; Glendenning et al., 2011; Abu-Libdeh et al., 2011; Shafaat et al., 2012). These overlays create self-contained replication groups of nodes which act as single nodes in the DHT. Routing is performed at the group level and not at the physical node level, allowing the system to fine-tune the replication sets. So, even though they are based on consistent hashing (and hence metadata-less approaches), these overlays allow decoupling the management of replication from management of the overlay topology. In Group-based DHTs, each node is a logical entity, materialized by a replica group of variable size. Contrary to classical DHTs, where each node has a pre-determined location in the network (depending on its identifier), in these approaches, nodes can join any existing replica group. All members of each group coordinate to act as a single node in a higher layer, defining a DHT. Since replication is decoupled from the DHT layer, the replication degree of individual groups can vary without affecting the DHT's structure. As a result, consistent hashing is used to place data items into groups, and data is then replicated among all nodes that belong a group. Rollerchain provides an alternative implementation of this approach.

3.1.2 Load Balancing on DHTs

Our work leverages on data replication to improve load balancing. One fundamental problem in DHTs is that node identifiers or, most likely, keys, or the load that each of them represents, may not be uniformly distributed in the address space. As a result, some nodes will be required to maintain (and answer queries for) many items while others may be relatively offloaded. Virtual servers (Dabek et al., 2001; Godfrey et al., 2004) are a common technique to circumvent this problem. Since each physical node joins the DHT using multiple identities, nodes may disable and enable their identities as needed to balance the load. On the other hand, having multiple virtual identities requires each node to maintain more routing information and monitoring more overlay neighbors, which may impose an excessive overhead. Also, this strategy amplifies the effect of churn, as the departure of a single physical node causes the simultaneous failure of multiple virtual servers.

Other approaches rely on making a guided choice of node identifiers at join time, to select positions in the identifier ring such that the load is evenly distributed among all nodes. In order to achieve this, works such as Kenthapadi and Manku, 2005 and Ledlie and Seltzer, 2005 use probes in the system to determine the best identifier to use. These schemes allow balancing the load of object storage among all nodes in the system without requiring additional routing information, by increasing the cost of join operations. However, they may create a non-uniform distribution of nodes in the identifier space, which hinders the performance of some routing algorithms.

3.1.3 Combining Structured and Unstructured Overlays

Our solution relies on the combination of structured and gossip-based overlay mechanisms. Some previous systems have already explored this idea, although with different goals and, as a result, with solutions that are structurally quite different from Rollerchain. The work by Ghodsi et al., 2007b makes the case for combining gossip-based and structured networks, and discusses several examples of successful synergies between both designs. This work claims that through this symbiosis, the current state-of-the-art on DHTs can be improved with new overlay designs that offer better reliability, lower bandwidth costs, or better geometry. Inspired by the work of Jelasity et al., 2004, Maniymaran et al., 2007 present an approach that follows this design principle and combines two overlays, a DHT and a interest-based unstructured overlay, at a cost similar to that of building a single one. Kelips (Gupta et al., 2003) is a DHT structured using virtual nodes composed of several physical peers. Unlike Rollerchain, each of these nodes know at least one contact in every other virtual node, creating a one-hop DHT. Kelips supports efficient lookups, but it does not present any solution for data replication and each node stores a pointer to every object owned by its virtual node, a property that severely limits its scalability.

3.2 Rollerchain Overview and Building Blocks

Rollerchain dynamically manages the replication groups by combining features of unstructured and structured overlay networks in an integrated design. More specifically, Rollerchain builds a DHT where each (virtual) node is materialized by a small group of peers, the size of which depends on the replication degree, R. These groups are neither static nor defined a priori. Instead, they are dynamically created and maintained by the unstructured component. Acting collaboratively, peers on each virtual node share among themselves the information required to maintain the DHT topology and the data stored by their virtual node. The unstructured component of Rollerchain is responsible for creating and maintaining the virtual nodes. Some of its mechanisms are inspired by Overnesia, an unstructured overlay that aggregates peers in clusters (Leitão and Rodrigues, 2014). The structured component of Rollerchain runs the DHT maintenance algorithms. For self-containment, before describing the operation of Rollerchain in detail, we provide a brief overview of Overnesia and of the Chord DHT (Stoica et al., 2001), whose architectures have inspired our design.

Overnesia Overnesia (Leitão and Rodrigues, 2014) is an unstructured overlay network, where nodes self-organize into fully connected clusters which, in turn, are highly and randomly connected among themselves. The target size of these clusters can be configured by the application to a given *targetClusterSize* value. However, the cost of ensuring that every generated cluster produced by the protocol has exactly the same size in highly dynamic and open environments can be prohibitively high. To overcome this challenge, Overnesia instead ensures that the size of clusters is distributed between *minClusterSize* and *maxClusterSize*, with a predominance of clusters with the *targetClusterSize* (evidently, *minClusterSize* < *targetClusterSize* < *maxClusterSize*). Each Overnesia cluster is assigned a random identifier that is known by all elements of that cluster. A gossip-based anti-entropy mechanism, in which elements of each cluster periodically and randomly exchange messages among themselves, is employed to ensure that cluster members converge on a consistent view of the cluster membership, despite concurrent joins and failures in the system. Furthermore, this process is used to provide a minimal amount of coordination required to increase the diversity of external links maintained by different cluster members. Note that Overnesia does not offer any DHT support. As a result, the way links are established among clusters does not take into account routing requirements, other than attempting to maximize the connectivity of the network.

Chord Chord (Stoica et al., 2001) is a widely known DHT that organizes nodes in a ring-like topology. It places objects in specific nodes of the ring using consistent hashing. Chord's ring is created by sorting nodes by their identifier modulus the size of the identifier space. Its maintenance is mostly proactive, such that each node keeps a predecessor and a successor node through periodic maintenance routines. More specifically, each node N periodically queries its successor S in order to obtain the predecessor P of S (naturally, in a stable scenario, we should have $P = N$); should $P \neq N$ be a node with an identifier in the interval $]N\_id; S\_id[$, N will then switch its successor pointer to node P. After this update, N informs P that it is now P's predecessor. This simple routine allows the ring to converge and remain connected even when facing concurrent entry and departure of nodes. Even though Chord's ring would suffice for any node to reach any other node in the overlay, routing messages exclusively through this ring would be very inefficient. Thus, Chord's protocol includes an efficient routing mechanism: each Chord node maintains a *Finger Table*, from which it selects the closest node on the ring to route messages towards their destination. As the Finger Table contains pointers to nodes which are at exponentially increasing distances from the node's position in the ring, this mechanism allows Chord to route messages in $log(N)$ network hops, where N is the total number of nodes (since the overlay's distance to the destination can be halved with each hop).

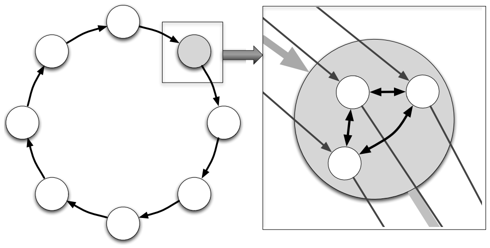


Figure 3.1: Rollerchain's Architecture: At the DHT level vnodes form a ring. Each vnode is composed of several pnodes.

3.3 Rollerchain Operation

3.3.1 Definitions and Basic Operation

We opted to preserve the nomenclature of the original Chord paper, where each peer is denoted a *node*. Therefore, in Rollerchain, each peer is called a physical node, or *pnode* for short. The unstructured component of Rollerchain aggregates pnodes in clusters. All pnodes that belong to the same cluster cooperate to behave collectively as a logical virtual node, or *vnode*. A *vnode* is a fully connected cluster of pnodes with a fluctuating size around R (the replication factor) that act as a single node in the structured layer. To facilitate the coordination among pnodes in the same vnode, an anti-entropy gossip-based protocol is executed among them. This protocol allows pnodes to exchange information required for the operation of Rollerchain (which we will incrementally describe), including the membership of the vnode and key/value pairs maintained by its members. Also, to simplify coordination among vnode members in several Rollerchain procedures, the member with the lowest node identifier in each vnode acts as the *vnode leader*. Occasionally, it may happen that more than a single pnode sees itself as the vnode leader. This does not affect the correctness of Rollerchain, as the leader is only used to reduce the signalling costs of the algorithms. Similarly, if no pnode sees itself as the leader, this only delays the progress of the algorithms until the anti-entropy procedure enables one of the members to see itself as the leader. Virtual nodes establish *virtual links* among themselves to create a logical ring. A virtual link between vnode A and B is materialized by establishing links among pairs of pnodes (a_i, b_j) where $a_i \in A \land b_j \in B$. The algorithm for establishing and maintaining virtual links is described later in the text. Figure 3.1 provides a visual representation of Rollerchain's architecture.

3.3.2 The Unstructured Layer

The unstructured layer of Rollerchain is an overlay composed of virtual nodes, where each vnode stores key/value pairs replicated by all of its members. This replication not only increases the resilience of data, but also allows pnodes to share the load of answering queries for objects stored in the associated vnode. When joining the unstructured layer, a pnode starts a random walk in the network, which probes suitable vnode to join. There is no restriction as to what criteria may be used to select the vnode to join. As an example, the new pnode may choose the most heavily loaded virtual node found in the random walk to join the network, in order to achieve good load balancing. If the load of a vnode is defined as the number of key/value pairs it stores normalized to the number of pnodes it is composed of, this mechanism causes heavily loaded virtual nodes to attract new members in order to share their load.

When a vnode leader detects that its vnode has become too large, it starts a division procedure to halve the vnode into two others with half the size. This division serves not only to reduce the costs of replicating data among its members, but also to reduce the number of objects each member stores. When dividing, the position of the new vnode in the identifier ring is critical for the good performance of the system. The new vnode may select a new identifier at will. However, if it just selects a new random identifier as in Overnesia, it will discard all of its data, rejoin the DHT and receive new data from its new successor. Thus, as a way to improve the efficiency of the structured layer, one of the newly generated vnodes may keep the identifier of the original vnode (to avoid inducing artificial churn in the structured overlay) and the other generates an identifier that allows it to become the owner of half of the original vnode's key/value pairs.

A vnode merge occurs when the vnode leader detects that its vnode's size is close to the minimum replication degree. The merge prevents objects from being lost, by integrating the vnode with its successor. The vnode leader obtains its successor vnode's membership and broadcasts it to its neighbors, which then change their vnode identifier.

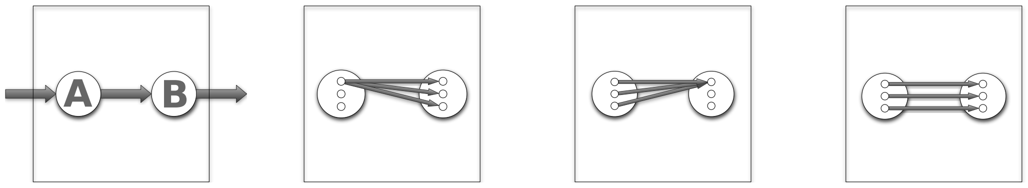
3.3.3 The Structured Layer

The structured layer of Rollerchain is a double-linked ring composed of virtual nodes provided by the unstructured layer. As described in the previous section, the vnode identifiers may be selected according to different policies. Therefore, the identifiers may not be uniformly distributed in the identifier space. In typical DHTs (such as Chord (Stoica et al., 2001), Pastry (Rowstron and Druschel, 2001b), or Kademlia (Maymounkov and Mazières, 2002)), node identifiers are assumed to be uniformly distributed in the identifier space. Thus, typical DHT routing mechanisms cannot be applied to Rollerchain. To ensure efficient routing in Rollerchain's identifier space under these conditions, each vnode in Rollerchain has a virtual link to its immediate successors and one finger table with $log(N)$ rows, containing virtual links to distant vnodes in the ring. As in DHTs such as Chord (Stoica et al., 2001) or Viceroy (Malkhi et al., 2002), each row in the routing table represents an exponentially larger jump than the previous row. However, whereas in these solutions the larger jumps refer to the *identifier space*, in Rollerchain each row in the routing table represents an exponentially larger jump in the *number of virtual nodes on the ring*. When vnodes are distributed uniformly in the ring, both schemes generate similar routing tables. However, if some areas of the identifier space have more vnodes, these will appear more frequently in Rollerchain's routing tables, whereas in typical DHTs, they would have the same probability, regardless of the density of those regions. It is important to notice that, even though this routing scheme ensures that our system can route messages logarithmically with the number of vnodes, other routing mechanisms, such as those based on Skipnets (Harvey et al., 2003) or the work presented in (Klemm et al., 2007), could also be employed.

3.3.4 Layer Interoperation

3.3.4.1 Virtual Link Creation

As described before, virtual nodes cooperate to create a ring-based DHT. To preserve the logical ring, and to support efficient routing, each virtual node maintains a virtual link to other virtual nodes in the ring. More precisely, a virtual link needs to be maintained to each different virtual node in the finger table (including the successor). Virtual links between two vnodes, say V_A and V_B, need to be materialized by links between individual members of these vnodes (Figure 3.2a). For fault-tolerance and load balancing, these links should be distributed evenly among these members. For instance, consider a vnode V_A that has n members and a vnode V_B also with n members. It would be highly undesirable to have a singe pnode a_k from V_A to have n links to each member of V_B (Figure 3.2b). It would be equally undesirable if all nodes in V_A



(a) (b) (c) (d)

Figure 3.2: Management of virtual links among vnodes.

establish a link to the same pnode b_k in V_B (Figure 3.2c). The ideal solution would be to have each pnode $a_i \in V_A$ to have a link to a different pnode $b_i \in V_B$ (Figure 3.2d). In this way, each pnode would be required to maintain a single link and the virtual link would be materialized by n independent physical links. Additionally, this ensures that $n-1$ physical links between V_A and V_B would still be alive after a single pnode failure/departure.

In order to quickly achieve an even distribution of links among members of a vnode, the vnode leader (the pnode with the smallest identifier), say a_0, coordinates a procedure of virtual link creation. This procedure is used when a vnode leader updates the virtual links of its vnode, particularly during vnode division and vnode finger update. We assume that, when the virtual link creation procedure is invoked, a_0 is aware of at least one member $b_k \in V_B$. The leader a_0 starts by requesting b_k to return the current snapshot of V_B's membership; we recall that there is an anti-entropy procedure running inside each vnode that keeps such information up-to-date. Knowing the full membership of V_A and V_B, the leader a_0 assigns links among V_A and V_B in a round-robin manner, until all pnodes of V_A and V_B are connected to some pnode in the other vnode (note that Figure 3.2 is an simplification of reality, as V_A and V_B generally do not have the exact same number of pnodes). Finally, this mapping is propagated to all members of V_A. Each individual pnode a_k then initiates the establishment of link(s) to its corresponding pnode(s) in V_B.

3.3.4.2 Virtual Node Maintenance

The membership of a vnode may change as pnodes join and leave the overlay. As it is crucial for the connectivity of Rollerchain that its ring remains connected despite membership dynamics, such changes require adjustments to the link assignments that materialize the virtual links maintained by each vnode to its successor and predecessor nodes.[1] Consider, first, the result of a join operation on virtual node V_A by pnode a_k. The pnode may help in materializing the virtual links maintained by the virtual node V_A. To this end, and for each virtual link, pnode a_k will attempt to "alleviate" the load of some other members of V_A, by serving as endpoint of some of their links. This can be achieved as follows. The information exchanged in the anti-entropy protocol includes the number of forward and backward links maintained by each pnode for a given virtual link. Therefore, a_k can select a pnode that has more forward or backward links than the majority of the remaining pnodes and connects to one of their endpoints. When the other pnode finds out about a_k's links, it closes the redundant link. On the other hand, if a_k finds that the number of links is perfectly balanced, the new pnode just creates a redundant link at random. Consider now the case where a pnode leaves a vnode. The physical links established by that pnode will break. This means that pnodes on the other endpoint of those links will perform the link re-distribution described in the previous paragraph. Finally, consider the case where multiple joins and departures occur in succession. If joins and leaves are perfectly interleaved, a (physical) link is lost in each leave but a new link is created in each join, and the number of links that materializes the virtual link between two vnodes remains constant. However, if bursts of leaves or joins occur, the balance may no longer be preserved. When, as a result of the anti-entropy procedure, the vnode leader detects heavy imbalance in the link distribution for some virtual link, it triggers a *re-balance* procedure. This rebalance procedure consists of sending a balanced mapping of physical links for the virtual link to the pnodes causing the imbalance, so that they may adjust their links accordingly.

3.3.4.3 Virtual Node Division

Vnode division can be performed in such way that the two new resulting vnodes have similar size and load. For this purpose, one of the vnodes preserves the identifier of the original vnode and the other vnode becomes its predecessor, by assuming an identifier that causes it to become the owner of half of the key/value pairs of the original vnode. This allows Rollerchain to dynamically adapt to the distribution of keys in the identifier space. Independently of the applied policy, virtual node division is controlled by the leader of the original vnode, that sends a message to all members with the identifiers of the new vnode, the membership of each vnode,

[1]Note that there is no need to adjust the remaining virtual links in the finger table, as those are not essential for maintain overlay connectivity.

and a new assignment for the virtual links maintained between them. This particular division procedure ensures that the new vnodes are neighbors in the DHT, contributing to savings in the number of keys moved during this operation, as nodes will only have to locally discard object/key pairs.

3.3.4.4 Virtual Node Merge

In a steady-state scenario, vnodes are stable, as new pnodes may replace failed/departed pnodes. However, as a result of multiple departures and failures, the size of a vnode may become below the desired replication level for the key/value pairs. When a vnode size becomes too small, it is merged with its successor. This ensures that all nodes that are part of the two merging vnodes retain their key/value pairs. The anti-entropy in the resulting vnode will ensure that those keys will be later replicated by all remaining members. Similarly to other mechanisms, the merge is started by the leader of the merging vnode. In highly dynamic environments, different nodes may see themselves as the leaders of the same vnode, and send out conflicting orders regarding vnode division or merge. However, the system is designed to tolerate these scenarios, as a merge order always supersedes a division order. Thus, even though such orders may create temporary inconsistencies (which are unavoidable in large asynchronous systems), the topology will eventually converge to a correct configuration.

3.3.5 Key/Value Pairs Replication

The replication of key/value pairs among the members of a vnode is performed using a combination of eager and lazy replication schemes. When a key/value pair is inserted in a vnode, eager replication is used. The pnode that receives the request uses a best effort application-level multicast primitive to replicate the pair among the other members of its vnode. Subsequently, replicas are maintained using a lazy replication scheme that leverages on the anti-entropy protocols executed among vnode members. Processes include a hash of the keys they own for their current interval of responsibility in the anti-entropy messages. If (as a result of an anti-entropy exchange) a pnode discovers that its hash differs from that of a neighbor, they exchange their full list of keys, so that both processes may request their missing key/value pairs from each other. To maintain the correct keys at each vnode, each pnode periodically checks the keys it stores. When a pnode detects it is storing objects for which the keys should be owned by its successor (the keys are between its vnode identifier and its successor vnode identifier) or its predecessor (the keys are not between its identifier and its predecessor identifier), it transfers the content to the correct vnode and deletes it locally. The data is sent to a pnode in the other vnode, which in turn sends the hash of its keys to all the remaining elements of its vnode, so that they may request their missing key/value pairs (if any). As with other replication mechanisms for which there is no master copy of the data (e.g. Ghodsi et al., 2007a), this mechanism does not provide strong consistency among the replicas. The anti-entropy mechanism running in each vnode guarantees that eventually all nodes will locally store all the key/value pairs.

3.3.6 DHT routing

DHT routing in Rollerchain is similar to Chord routing with some twists. Logically, lookups follow virtual nodes, using the virtual links maintained in the vnode's finger table. In reality, lookups are implemented by individual pnodes. When the lookup arrives at a pnode, it uses its own finger table to select a pnode to be the next hop for the lookup. If the pnode cannot contact the next hop, the lookup is re-routed to another pnode in its own vnode, which attempts to forward the lookup using one of its own links. This is similar to the re-route mechanisms used by DHTs that have routing tables with alternate paths (for instance, Pastry (Rowstron and Druschel, 2001b)). Due to the redundancy in the virtual link maintenance, it becomes very hard to undermine routing in Rollerchain. As in Chord, all lookups end in the predecessor of the key, which return their successor as the owner of the key. In Rollerchain, when lookups reach one pnode in the predecessor vnode of the target key, it returns a random successor pnode (we recall that each pnode knows the composition of its successor vnode through the anti-entropy protocol). This allows the query load to be equally distributed by all nodes despite some (possibly temporary) imbalance in incoming links.

3.4 Topology Evaluation

Rollerchain allows improved data placement through the previously described mechanisms, but, nonetheless, these gains are not achieved by sacrificing or trading off fault-tolerance. To study the fault-tolerance of Rollerchain, we have conducted experiments to compare it with other state of the art replication mechanisms on a (highly) dynamic scenario. All results reported in

Table 3.1: Percentage of objects reachable at the end of the simulations.

	$c = 1$	$c = 10$	$c = 100$
Rollerchain	100.0%	100.0%	98.2%
Neighbor Replication	100.0%	94.1%	0.0%
Multi-Publication	100.0%	10.2%	0.0%

this section were obtained using the Peersim simulator (Montresor and Jelasity, 2009) event-based engine. The system was populated with 50,000 objects and is composed of 10,000 nodes. To extract comparative measures, we used Chord with the Neighbor Replication scheme and Chord with Multi-Publication (as described earlier).

3.4.1 Experimental Parameters

All Chord configurations used a replication factor of 7 (in Neighbor Replication, each key is replicated in the owner and in 6 of its successors). Rollerchain was configured with a *minClusterSize* of 3 and a *maxClusterSize* of 8, which (we experimentally determined) results in vnodes being composed, on average, by 7 pnodes. This means the average replication factor of all replication techniques is the same. Furthermore, Rollerchain's routing table size was set to 11, to ensure that both protocols have a similar number of (distinct) fingers, creating routes with the same number of hops (15 on average). The anti-entropy mechanism of Rollerchain was also executed as often as the replication maintenance routines of the other replication schemes. The results presented are the average of, at least, 5 individual simulations for each scenario.

3.4.2 Fault-Tolerance Comparison

In order to evaluate the availability of the three solutions, we have conducted experiments as follows. All overlays were initialized by having nodes join the system one at a time. After a stabilization period, churn was induced for 10,000 consecutive simulation steps (one step corresponds to up to 5 Round Trip Times). In each step, c nodes were concurrently removed and c new nodes were added (c is dubbed *churn rate*). We performed experiments with churn rates of 1, 10, and 100. By using increasing churn rates, we can assess how each overlay and each replication scheme responds to increasing dynamics in system membership.

Table 3.1 shows the percentage of data items reachable at the end of the simulations for the same churn rate. For the smallest churn rate, all solutions retain all items. However, for higher churn rates, frequent changes in the network topology cause havoc in the replica maintenance schemes of Neighbor Replication and Multi-Publication, due to the increasing number of routing inconsistencies. Since Multi-Publication's replica maintenance mechanisms highly depend on being able to route requests on the overlay, this replication scheme loses a large percentage of the data items even for the intermediate rate of churn ($c = 10$). As c is increased, Neighbor Replication also fails more frequently and for the highest rate of churn all items are lost due to the overlay becoming partitioned. Rollerchain shows a more robust behaviour, due to the fact that for all rates of churn but the highest, the structured layer remains mostly unchanged. Under heavy churn, we observe some minimal data loss, related with the unavoidable but sporadic simultaneous failure of whole replication groups.

Overlay maintenance costs

Figures 3.3 and 3.5 show the bandwidth consumption resulting from the sources of overhead: *i*) the signalling costs of the monitoring protocols that are required to preserve the replication degree (Figure 3.3); and *ii*) the costs associated with maintaining the overlay topology (Figure 3.5).

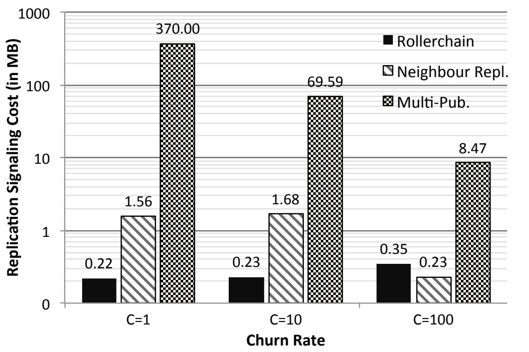


Figure 3.3: Replication signalling costs per node, considering 128bit keys.

The costs of maintaining replication in Rollerchain are not significantly different from those of Neighbor Replication. In both protocols, each node has to gossip with a limited number of neighbors (one in the case of Rollerchain, as each pnode gossips with only one other neighbor on each round, and $R-1$ in the case of Neighbor Replication). Also, in both cases, the message contains only a simple hash of the node's data. On the other hand, Multi-Publication's costs are at least two orders of magnitude higher. This results from the periodic lookups that a node is

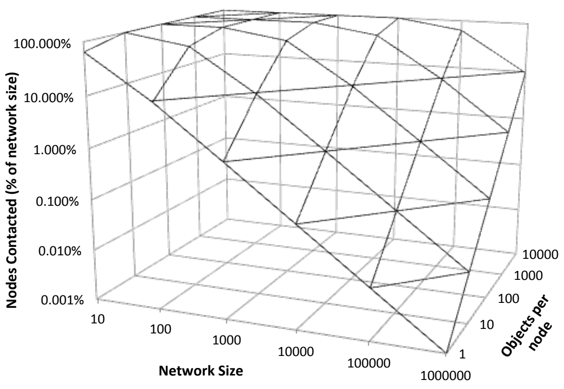


Figure 3.4: Average number of nodes contacted by nodes running multi-publication, considering $R = 10$.

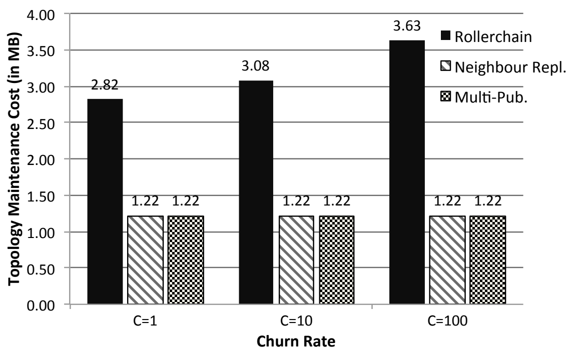


Figure 3.5: Topology signalling costs per node.

required to perform for every key it owns. These lookups are routed in the overlay and incur in multiple message exchanges. In fact, when considering only the periodic replication mechanisms, Multi-Publication requires a node to monitor $R - 1$ nodes in the overlay for each key it stores. Since the replica nodes for each object are distributed in the overlay using consistent hashing, the larger the system is, the less likely it is that different objects are replicated in the same node. Thus, even in stable topologies (with no node joins or leaves), the number of neighbors each node has to contact increases with the network size, the number of objects, and their replication degree. This property severely limits the scalability of Multi-Publication (as Figure 3.4 shows).

When considering the topology maintenance overhead (Figure 3.5), it can be observed that Rollerchain is more expensive than the other two techniques. Those results are not surprising, as Rollerchain combines design elements of two types of overlays. Still, these costs are in the same order of magnitude of competing approaches, and relatively small, when compared with the costs of replicating the data itself. For example, considering a churn rate of $c = 1$, if objects have a size of 5MB (the typical size for music audio files), each node in Rollerchain will require 190MB of bandwidth to move objects; whereas each node using Neighbor Replication would consume approximately 385MB of bandwidth. In fact, for a system with the number of objects and replication degree considered in this section (resp., $50,000$ and 7), the overall costs of Rollerchain are below those of Neighbor Replication for object sizes greater than 6.7KB. Finally, the increase in bandwidth consumption by Rollerchain under higher churn rates can be explained by the occurrence of merge and division operations, which require additional coordination among peers, but also make the operation of the overlay more robust as discussed earlier.

3.5 Replication Policies

In order to demonstrate Rollerchain's improved flexibility, we propose several replication policies that take advantage of this design to improve various parameters of the overlay, such as monitoring costs, replication costs, and load imbalance costs. These policies use information on node reliability, group size, and group load, in order to select which groups are most adequate for each new node to join.

Given the relevance of implementing data replication in P2P systems, this topic has been extensively studied in the literature. Surprisingly, despite all the results achieved so far, to maintain replicated data in this setting in an efficient manner still remains a significant challenge. In fact, in a talk at Middleware 2011, Druschel and Rowstron have identified data replication as one of the problems for which no satisfactory solution had been proposed yet.

What makes data replication particularly challenging is that maintaining the replication degree incurs several costs, and the goal of reducing the costs in one dimension of the problem typically conflicts with the goal of reducing the costs in the remaining dimensions. Namely, in this work we concentrate on the following 3 different relevant costs metrics of a data replication scheme:

- *Monitoring costs:* the costs associated with monitoring existing replicas to assess if they are still live, such that new replicas are spawned timely to replace failed replicas. This is required to preserve the desired replication degree and ensure that data is not lost.

- *Data transfer costs:* the costs associated with the creation of new replicas in the system. Blake and Rodrigues, 2003 have shown that data transfer costs account for a significant part of the bandwidth consumption in the system supporting replication and that these costs effectively limit the amount of data that can be stored.

- *Load imbalance costs:* some replication policies may result in an imbalanced replica distribution among existing nodes. Load imbalance has a detrimental effect on the operation of the system, since some nodes will be overloaded while the capacity of other nodes will be underused.

As hinted above, to optimize all these costs simultaneously may be impossible. For instance, some replication policies favor the placement of replicas on nodes that are known to be more reliable. This allows for saving in data transfer costs (since the creation of new replicas becomes less frequent) at the cost of introducing load imbalance (the more reliable replicas become overloaded). Furthermore, how these tradeoffs are tackled by a specific algorithm is often hard to infer, as proposers of a given replication scheme typically evaluate their solution using only a subset, if not only one, of the metrics above, neglecting the impact of the proposed policy on the other dimensions of the problem. Considering these observations, we:

i) provide a comparative study of the most relevant data replication policies that have been proposed in the literature, highlighting the tradeoffs they implement when considering the different costs involved.

ii) propose a number of novel data replication policies based on virtual node architectures such as that of Rollerchain, that have not been previously experimented in the literature.

iii) show that some of the novel policies outperform previous strategies. In particular, one of these policies, when compared with other competing solutions, achieves large bandwidth savings, offers good load balancing, and has no negative impact on the monitoring costs. The best policy is based on a strategy that may appear counter-intuitive at first sight: it uses the less reliable nodes to store the most accessed data items. The rationale for the success of this strategy will be clear later in the text.

3.6 Tradeoffs in Data Replication

The reader may find it surprising that, given the huge body of research in data replication in the context of P2P systems (Rowstron and Druschel, 2001b; Glendenning et al., 2011; Stoica et al., 2001; Dabek et al., 2001), this is still an open topic. However, to the best of our knowledge,

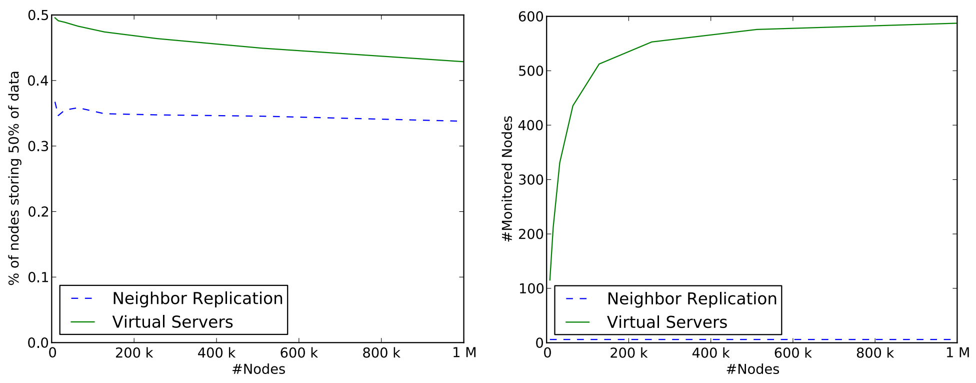


Figure 3.6: Load balancing vs monitoring costs tradeoff

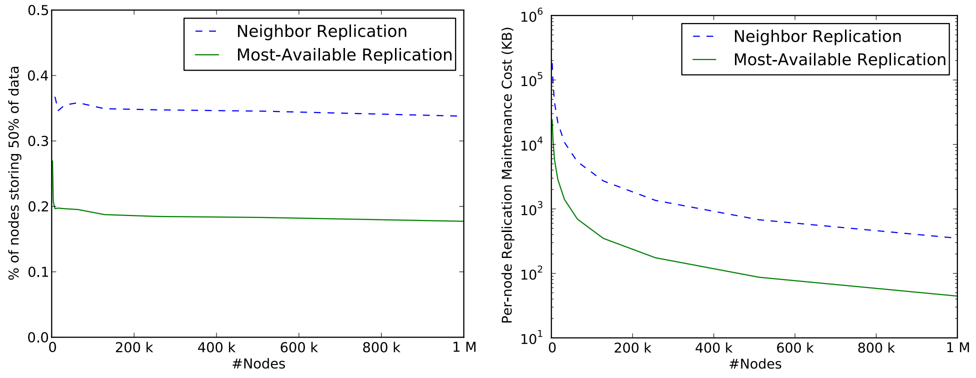


Figure 3.7: Load balancing vs data transfer costs tradeoffs

no studies have been published addressing the costs of data replication on its multiple dimensions. The work of Blake and Rodrigues, 2003 has focused on the data transfer costs and its impact on the scalability of the system. The work by Pace et al., 2011 has addressed data transfer costs and load imbalance, while the work of Ghodsi et al., 2007a focused mostly on monitoring costs. As a result, it remains particularly difficult to understand the tradeoffs involved in the different replication policies that have been proposed in the literature.

To motivate the need for new replication policies, we illustrate our point using 3 well-known replication policies: *Neighbor Replication (Stoica et al., 2001; Rowstron and Druschel, 2001b)*, *Neighbor replication with virtual servers (Dabek et al., 2001; Godfrey et al., 2004)*, and *Most-available node replication (Mickens and Noble, 2006)*, the previously presented policy which places replicas in the nodes that are less likely to fail or to leave the overlay.

Figures 3.6 and 3.7 show the comparative performance of these policies according to different metrics (we will postpone a detailed description of the experimental setting to Section 3.9, where we do a more detailed analysis of these and other replication policies). Figure 3.6 illustrates the tradeoff between the load imbalance costs and the monitoring costs of the first two policies above. The introduction of virtual servers augments the number of (virtual) nodes in the overlay and, therefore, promotes a good load balancing among the replicas. These advantages are well documented in the literature (Dabek et al., 2001; Godfrey et al., 2004). On the other hand, each node has to keep track of a different set of neighbors for each of its identities. Therefore, the improvements in load balancing come at the cost of a proportional increase in the monitoring costs. Figure 3.7 illustrates the tradeoff between load unbalace costs and the data transfer costs, when comparing Neighbor Replication *vs* Most-available node replication. By selecting the most reliable nodes to store replicas, fewer items are affected by failures and fewer replicas need to be respawned. This introduces significant savings in the data transfer costs, as reported in Mickens and Noble, 2006, Pace et al., 2011. Unfortunately, these gains come at the cost of unbalancing the load in the system. In fact, we have observed that the load imbalance can be 2000 times worse than other state-of-the-art solutions (see Section 3.9).

The examples presented before provide an insight on the problems addressed in the current work. Namely, we address the following questions: Do other previously proposed policies implement similar tradeoffs? Are there replication policies that can reduce the costs in a given metric without significantly increasing the costs in another metric? Are there unexplored policies that can implement more advantageous tradeoffs? Is it possible to define a framework that helps in comparing the performance of different policies?

To answer these questions, we will first classify the main replication policies that have been proposed in the literature, then propose some novel policies, and finally provide a detailed comparative analysis of the different alternatives.

3.7 A Catalog of Previous and New Policies

In this section, we start by introducing a classification that helps in comparing the policies for data replication according to the their principles of operation. Subsequently, using this classification, we list the most relevant approaches that have been proposed in the literature. Finally, we propose some novel policies that, to the best of our knowledge, have not been presented before.

3.7.1 Policy Classification

We first distinguish *oblivious* from *informed* policies. Oblivious policies do not take into consideration the properties or state of each peer and consider only topology properties of the overlay. For instance, neighbor replication, introduced before, fits in this category. On the contrary, informed policies collect information about each individual node, such as the expected availability, or the current load, to make decisions about data placement. For instance, the Most-available policy fits in this category.

Another important dimension that can be used to classify replication policies considers the machinery required to locate replicas. Here, we distinguish two opposing strategies: *consistent hashing* and *directory-based* lookups. When consistent hashing is used, replica location is derived by the hashing function. As a result, the policy has little control over data placement but, on the other hand, replicas can be located in a very efficient manner. Directory-based approaches can place replicas in arbitrary locations but require a directory lookup to find the replica's location; this may involve one or more round-trips in the network.

Finally, we can distinguish policies according to the requirements they impose on the underlying overlays. This work considers exclusively replication policies for structured overlays that implement some form of DHT. However, besides traditional DHTs such as Chord (Stoica et al., 2001) or Pastry (Rowstron and Druschel, 2001b), we also consider overlays that use virtual servers (Dabek et al., 2001) and logical groups. Virtual servers will be introduced in Section 3.7.2. Logical groups refers to recently proposed overlays that take a different approach to replication, by creating self-contained replication groups of nodes which act as single nodes in the DHT. Rollerchain is an example of an implementation of such overlays.

In Table 3.2 we consolidate the different criteria together. In the next paragraphs we discuss how they can be combined to build different strategies:

- *Strategies based on consistent hashing:* The strategies rely on node (or group) identifiers and on randomization to place replicas. In this case, virtual servers only offer better randomization, because the number of node identifiers is larger. If the DHT is not group-based, there are little opportunities to improve the operation of the network, other than carefully selecting the node identifiers, to compensate for load imbalance. In the case of group-based DHTs, it is also possible to change the size of the groups, as nodes join and leave, or even by migrating nodes

Table 3.2: Design Space of Replication Policies for P2P Systems

	consistent hashing			**directory based**		
	plain	**virtual servers**	**groups**	**plain**	**virtual servers**	**groups**
oblivious	baseline	achieve better randomization	manage group size to avoid churn at the logical level	avoid migrating replicas		no advantage
informed	change node ids to distribute keys		change group ids and place nodes in the right groups	place replicas in better nodes		

from one group to another.

- *Directory-based strategies:* Oblivious strategies use directories to place replicas randomly but can maintain the replicas in the same nodes, regardless of their position in the DHT, to avoid moving replicas. Informed strategies can do better, by placing replicas taking into consideration the properties of nodes. However, directories bring no advantages when group-based DHTs are used, because groups have no intrinsic characteristics: the characteristics of each group can be tuned at will by the policy, as described before.

3.7.2 Previous Replication Policies

We now identify and describe the most relevant policies that have been previously proposed in the literature.

3.7.2.1 Strategies based on consistent hashing

Oblivious Strategies

We consider the following three oblivious strategies that rely on consistent hashing: neighbor replication, multi-publication, and neighbor replication with virtual servers.

Neighbor Replication (NR): As previously described, Neighbor replication (Stoica et al., 2001; Rowstron and Druschel, 2001b) keeps copies of each key-value pair in the r neighbors of the node responsible for the key. This policy keeps a tight control on replication degree, which, as demonstrated in Blake and Rodrigues, 2003, has a high replica maintenance cost, since replicas must be created and destroyed with every change in the network. However, given the fact that each node replicates in a fixed set of neighbors, which is independent of how many data items are stored in the system, this solution has low monitoring costs.

Multi-Publication (MP): Multi-Publication (Ratnasamy et al., 2001; Knezevic et al., 2005) stores r replicas of each data item in different and deterministically correlated positions of the DHT. Multi-Publication offers very good load balancing properties, but can have expensive monitoring, since it uses DHT routing to keep replication of each item it stores. As previously shown, in the worst case, a node that is the owner of M objects, each replicated in r other locations, has to periodically monitor $M \times r$ different nodes.

Neighbor Replication with Virtual Servers (NR+VS): Virtual servers (Dabek et al., 2001; Godfrey et al., 2004) are a common technique to circumvent the load imbalance problem. However, the efficiency of this mechanism depends on how many virtual servers each node can handle, but having a larger number of virtual servers increases the monitoring costs.

Informed Strategies

To the best of our knowledge, the only informed strategy that does not require the use of a directory has been proposed in the context of group-based DHTs in Glendenning et al., 2011 (the name has been given by us, as the original Scatter paper provides no names to the proposed policy).

Resilient Load-Balancing (R-LB): This policy uses the underlying overlay mechanisms to address resiliency and load-balancing. Resiliency is tackled by having nodes join the groups with fewest members and by merging a small group with its successor. Furthermore, the policy attempts to balance the load of each individual node, by making sure that when a group is split into two, each new group will process a load proportional to its size. To implement this policy, the overlay mechanisms are configured as follows: new nodes join the group with the highest *per-node* load; group identifiers (and, therefore, key assignment) remain unchanged until a group needs to be split or merged; when the group splits, keys are divided such that the per-node load of each resulting group is balanced (i.e, the identifier of the new group is chosen so that it owns a portion of the load proportional to its number of members).

3.7.2.2 Directory-based strategies

Oblivious Strategies

We consider the following oblivious strategy that requires the use of a directory.

RelaxDHT: This policy, presented in Legtchenko et al., 2009, is a natural extension of neighbor replication. Initially, replicas are created in the r closest neighbor of the item owner. However, unlike neighbor replication, as new nodes join the neighbor set of the item owner, the constraint that replicas should be the closest nodes to the item owner is relaxed. Hence, replicas are not moved as new nodes join, and may drift away from the item owner. To lower the monitoring costs, the work in Legtchenko et al., 2009 imposes a limit on how farther away from the item owner replicas may drift. The main intuition behind this work is that by relaxing the topological constraints, one can achieve lower replica maintenance costs. However, the fact that nodes which are “older” in the system tend to store more replicas, leads to an imbalance in load.

Informed Strategies

We consider the following informed strategies: most-available and regularity-based.

Most-available: The work in Mickens and Noble, 2006 presents a policy which places data in the nodes predicted to be most available in the future. Similarly to the RelaxDHT policy, data is never relocated from a node as long as it is available. Hence, even though this technique can achieve particularly low replication maintenance costs, it leads to high monitoring costs.

Regularity-based: This policy, introduced in Pace et al., 2011, takes advantage of the fact that nodes may exhibit connection regularity. This means nodes connect to the system on regular patterns. Hence, the system can form groups of nodes which with a given probability will always be online to replicate the data. Unlike other works, this policy assumes nodes can keep persistent state between joining and leaving the system. The regularity-based policy biases replication towards a set of nodes (the most regular ones), which impacts negatively load balancing in the system.

3.7.3 Novel Policies

As mentioned in the previous section, Group-based DHTs open very interesting avenues to design novel policies for data replication in P2P systems and the work of Scatter has only explored the surface of these possibilities (in fact, in their paper, Glendenning et al., 2011 recognize the development of such policies as an intriguing direction for future work). In this work we follow that path by proposing a number of viable alternatives to the R-LB policy introduced in Glendenning et al., 2011.

3.7.3.1 Oblivious Strategies

We consider the following three oblivious strategies for group-based DHTs:

Random: This policy achieves load balancing through randomization. It works by letting nodes generate a random identifier and join the group responsible for that identifier. Similarly, when a group becomes too large, it divides, creating a new group with a random identifier which joins at a random location. When a node becomes too small, it disbands and all its members join in random locations. The goal of this policy is to serve as a baseline that can be used to assess the relative merit of other strategies for group-based topologies.

Supersize-me: This policy consists of tuning the overlay management mechanisms to keep groups much larger than the average replication degree which the policy is configured with. In this way, the amount of redundancy is increased but the likelihood that a group collapses due to the lack of replicas becomes very small. This may avoid some unnecessary data transfers in the system.

Avoid Surplus: This policy consists of the opposite of the “Supersize-me” policy above. The goal is to keep each group as close as possible to minimum replication degree, by having nodes join the largest groups such that those divide more frequently. This reduces the amount of redundancy in the system but creates the potential for better load balancing (more groups will exist in the overlay) and for reducing the monitoring costs (which is directly related to the group size).

3.7.3.2 Informed Strategies

We consider the following two oblivious strategies for group-based DHTs:

Preemptive Replacement:. The rationale for this policy is to adapt ideas that have been proposed for classical DHTs, such as the “Most-available” policy introduced in Section 3.7.2 to

Table 3.3: Policy Map

		Primary performance target			
		none	**monitoring**	**load balancing**	**bandwidth**
Oblivious	Plain		Neighbor Replication (NR)	Multi-Publication	RelaxDHT
	VServers			Neighbor Replication + Virtual Servers (NR+VS)	
	Groups				Supersize-me
Informed	Plain				Most-available, Regularity-based
	Groups	Random	Avoid-Surplus	R-LB	Preemptive replacement
			Hotter-On-Ephemeral		

group-based DHTs. The key idea is to rely on estimates of node-reliability to make new nodes join groups where existing nodes are most likely to fail, as a preemptive measure to avoid the group becoming too small and forced to execute a merge.

Hotter-On-Ephemeral (HonE): This policy aims to place the most used items (i.e., those which represent the highest load) on less reliable nodes. The key observation behind this policy is that the per-group distribution of keys of R-LB reduces load imbalance but ignores which nodes compose which groups. This insight, combined with the fact that the groups which store the top most-used keys will store fewer keys to achieve a balanced load, drives the design of Hotter-On-Ephemeral. Thus, Hotter-On-Ephemeral places the less reliable nodes in the groups which in R-LB store fewer keys, i.e., the groups that store the most-accessed keys in order to maintain the good control over load imbalance and the good monitoring costs of R-LB, while considerably reducing its bandwidth usage, since most joins will be performed in groups with fewer keys. In order to make sure that the groups remain stable, this policy also allows groups composed of mostly unreliable nodes to grow larger than the remaining ones, similarly to “Supersize-Me”.

3.7.4 Summary of All Policies

Table 3.3 provides a summary of all policies described in this section. Lines represent different techniques and columns the main performance criteria that is aimed by the policy.

Note that Hotter-On-Ephemeral aims at addressing the three performance criteria in a holistic manner.

3.8 A Performance Model To Compare Policies

We now introduce two metrics that can be used to compare the performance of different policies: the *message overhead* and the *imbalance ratio*. These two metrics are defined as a ratio between the performance of an actual system using a given policy against the performance of an abstract idealized system (referred to as the baseline system).

3.8.1 System Parameters

Our metrics rely on the following system parameters: *Number of Nodes (N)*, the average number of nodes in the system; *Number of Keys (K)*, the total number of keys stored in the system; *Replication Degree (R)*, the desired replication degree (each key should have R replicas); *Key Size (d)*, the amount of stored data associated with each key; *Churn Frequency (c)*, the number of joins and leaves per unit of time; *Monitoring Period (m)*, the number of units of time between two consecutive monitoring procedures; *Unit Monitoring Cost (UC_m)*, the cost of checking if another node is alive (typically, at least one "I'm alive" exchange); *Unit Transfer Cost*, ($UC_t(d)$): the cost of transferring one key from one node to another (depends on the data size d); *Load (L)*, the total number of requests the system has to satisfy per unit of time.

3.8.2 Baseline System

Using the above parameters, we define the following performance metric for an abstract idealized baseline system. The idealized system is completely homogeneous: the load is uniformly distributed among keys, such that if all nodes store exactly the same number of keys, then all nodes are subject to the same load. Furthermore, the probability of a node leaving the system is the same for every node and, when a node fails, its load and objects are scattered uniformly across all other nodes, such that these values are preserved. Also, in this system, nodes monitor the bare minimum number of neighbors for maintaining the target replication degree (R).

Baseline Average Number of Keys per Node (BK_{avg}): The average number of keys per node is $BK_{avg} = K \cdot R/N$.

Baseline Monitoring Cost (BC_m): We consider that a node should be responsible for at least one key, and therefore has to monitor at least R other nodes. Therefore $BC_m = R \cdot UC_m$.

Baseline Join/Leave Cost ($BC_{join\_leave}(d)$): We denote the baseline join/leave cost as the cost of transferring BK_{avg} keys to a node, i.e., $BC_{join\_leave} = BK_{avg} \cdot UC_t(d)$.

Baseline Load Balancing (BLB): In the idealized system, all nodes have the same load. We define the baseline load balancing as the fraction of system nodes that, together, satisfy half of the system load in this setting, which, by definition, is 0.5 (i.e., half of the nodes are required to serve half of the system load).

3.8.3 Actual System

The same metrics can be defined for a concrete system, using a particular policy. Namely:

Actual Average Number of Keys per Node (AK_{avg}): The actual average number of keys per node that results from applying a given policy.

Actual Monitoring Cost (AC_m): The actual (average) monitoring cost that results from applying a given policy. This depends on how many nodes each peer must keep monitoring.

Actual Join/Leave Cost ($AC_{join\_leave}(d)$) : The actual (average) join/leave cost that results from applying a given policy. This depends on how many data items need to be transferred upon each join and upon each leave.

Actual Load Balancing (ALB): The fraction of system nodes that, together, satisfy half of the system load. Let L_i be the number of requests per unit of time served by node i. Let $\mathcal{H} \subset \mathcal{N}$ be the smallest subset of system nodes such that $\sum_{i \in \mathcal{H}} L_i = L/2$. Then, $ALB = \frac{|\mathcal{H}|}{N}$.

3.8.4 Policy Efficiency

Finally, using the metrics above, we define the two criteria to measure the efficiency of a given policy. These metrics are defined as ratios between the performance of the actual system and the performance of the idealized system. Namely:

Message Overhead ($O_{message}(d)$): We define the message overhead of a given policy as the ratio between the actual and baseline values for the sum of the monitoring and join costs during a monitoring period:

$$O_{message}(d) = \frac{N \cdot AC_m + c \cdot m \cdot AC_{join\_leave}(d)}{N \cdot BC_m + c \cdot m \cdot BC_{join\_leave}(d)}$$

Imbalance Ratio (IR): We define the imbalance ratio of a given policy as the ratio between the baseline load balancing and the actual load balancing:

$$IR = \frac{BLB}{ALB}$$

In the next section, we will evaluate the different policies using the two metrics above. Note that, for both policy efficiency metrics, the higher the value, the less efficient is the policy with regard to the idealized baseline system.

3.9 Evaluation

In this section, we present experimental results that compare the performance of the replication policies presented in the previous section. To evaluate the policies, we have performed extensive simulations using the Peersim simulator (Montresor and Jelasity, 2009) cycle-based engine, running the different policies against similar workloads in internet-scale settings, as described below.

3.9.1 Simulation Settings

In order to simulate the real working conditions of an internet-scale deployment of a key-value store, we have used a real-world trace of connections and disconnections in a peer-to-peer network (Blond et al., 2009). The trace represents the activity of over 14 million unique users, of which we considered 1 million random users. We used one second of the trace as unit of time.

Table 3.4: Evaluation Parameters

N	15434
K	100000
R	6
c	3.3838
m	60
UC_m	84B
$UC_t(d)$	84B $+d$
L	1931660

We populated the system with 100.000 key-value pairs with a load following a Zipf distribution with $\alpha = 2.5$. For all group-based DHTs, we have configured the DHT with *min_group_size* and *max_group_size* 4 and 8, respectively. We experimentally determined this configuration to yield an average virtual node size of 6 for policies (except for Supersize-me and Avoid Surplus), which according to Glendenning et al., 2011 and Paiva et al., 2013 is enough to prevent data loss in most scenarios. For fairness, we also configured the remaining policies with replication degree 6. For the policies that allow the group size to surpass *max_group_size*, we used *max_group_size* $= 16$. The virtual servers' configuration used 100 virtual identities. Table 3.4 presents the remaining values for all parameters of the evaluation.

3.9.2 Results

The results from our experiments, namely the values for the message overhead (as a function of different item sizes) and the imbalance ratio, as defined in the previous section, are presented in Table 3.5. The table presents the values that have been experimentally measured using the simulation for all the policies. We remind the reader that, the higher the value, the poorer the policy's performance.

We start by discussing the results concerning the message overhead of the different policies. These results are summarized in the following list of observations:

- As expected, Neighbor Replication (NR) has 1.0 for the bandwidth, since it has the exact same behaviour of the idealized baseline system.

- Virtual Servers (VS) and Multi-Publication (MP) have a high impact on bandwidth, especially when d is small, since the monitoring messages represent an important part of the cost. When using VS, each node monitors on average 587 neighbors, while when using MP, each node

Table 3.5: Results for Evaluated Policies.

	$\boldsymbol{Omessage}(1KB)$	$\boldsymbol{Omessage}(1MB)$	$\boldsymbol{Omessage}(1GB)$	$\boldsymbol{IR}$
Neighbor (NR)	1.00	1.00	1.00	1774.1
NR+VS	46.73	1.27	1.18	114.5
Multi-Publication (MP)	17.09	1.21	1.18	1.5
RelaxDHT	0.11	0.21	0.21	2365.5
Most-available	0.07	0.13	0.13	2365.5
R-LB	0.71	0.52	0.52	1.1
Random	0.71	0.60	0.60	19.5
Avoid Surplus	0.76	0.41	0.41	308.5
Supersize-me	1.07	0.79	0.79	1.1
Preemptive	0.50	0.30	0.30	25.7
HonE	0.61	0.28	0.28	1.1

monitors on average 209 neighbors. In fact, however, should we consider a larger set of keys, MP would achieve an even worse result, since for each key a node owns, it potentially monitors R different neighbors.

- RelaxDHT and Most-available achieve the lowest bandwidth usage of all. This is expected, given that these strategies make sure that the nodes that fail the most do now own many keys. Interestingly, due to the fact that many nodes do not store any keys, these nodes are not required to monitor other neighbors, and for small values of d, these three strategies in fact achieve better $MessageOverhead$ results.

- Most of the group-based policies result in a bandwidth improvement, especially when d is larger. For small values of d, group-based policies have a similar degree of replication as the baseline, and thus achieve a similar monitoring cost. For large values of d, group-based policies benefit from the fact that they support variable replication degree. Thus, while the baseline solution (and NR) must move $BC_{join\_leave}(d)$ objects for each join and leave, group-based policies can allow the replication degree to decrease after a node leaves, and hence avoid moving $BC_{join\_leave}(d)$ objects for about half of c. This fact is clearly observable for R-LB, which achieves 50% less message overhead for the larger values of d.

- The group-based solution that obtains the best bandwidth usage is HonE, which, despite creating groups slightly larger than preemptive (noticeable by the higher message overhead for low d), can improve over the latter solution for larger key sizes. This is due to HonE actively forcing nodes to fail on groups with a smaller number of keys, while preemptive is only making sure that groups remain stable and failed nodes are most of the time replaced with new nodes.

- From the group-based policies, the Supersize-me policy achieves the worse results. Even though it does lead to a considerably smaller number of merges, in fact it forces nodes to transfer more data with each node join, since the groups being larger also causes each individual group to store more keys on average. On the other end, HonE benefits from the best bandwidth usage, due to avoiding group merges and directing ephemeral nodes to groups with fewer keys. In fact, for larger values of d, HonE can achieve a result comparable to that of the best overall solution (Most-available), with a message overhead little than $2\times$ worse.

We now analyze the results when considering the load imbalance that is caused by each policy. Again, we summarize our findings in a list of observations:

- All policies which are not designed to explicitly handle load present results considerably worse than those of the baseline. The most flagrant cases are for the RelaxDHT and Most-available policies, which are not only oblivious of the load each key represents, but also allow a large percentage of nodes to store 0 keys (up to 10% for Most-available).

- Virtual servers (VS) and Multi-Publication (MP) can obtain a low load imbalance ratio (despite the very negative effect on bandwidth/monitoring). MP achieves the best results, since it makes sure that the probability that two nodes store the same keys is low, while when using VS, it is common for two virtual servers to own the same data. Hence, should one VS have a load above or bellow average, this effect is amplified due to R other VS replicating its data.

- The Avoid Surplus policy results in a poor load imbalance ratio, as groups have no extra capacity to absorb failures, forcing merges that difficult the task of balancing the load.

- The Supersize-me policy is able to achieve a low load imbalance ratio. This is due to, in practice, its join and division operations being similar to those of R-LB, except for the fact that the size of the groups is allowed to fluctuate up to larger values. As a result, only the message overhead costs are negatively impacted.

- As expected, Preemptive and Random achieve poor results for load imbalance due to the lack of load balancing concerns. On the other hand, R-LB and HonE achieve similar results, both very close to the baseline, due to being designed with load balancing in mind.

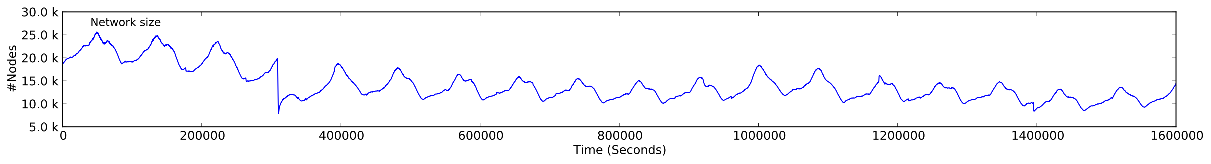
3.9.3 Detail

Figure 3.8 presents detailed views of how many keys are moved in the system as result of joins and merges, as well as the network size, as time progresses in the simulation. For clarity, we present only 0.1% randomly sampled points for join operations. The remaining points (for the network size and merges) are presented unsampled. We present detailed views only for the most interesting policies, HonE, R-LB, and Supersize-me. We highlight the following aspects of these solutions:

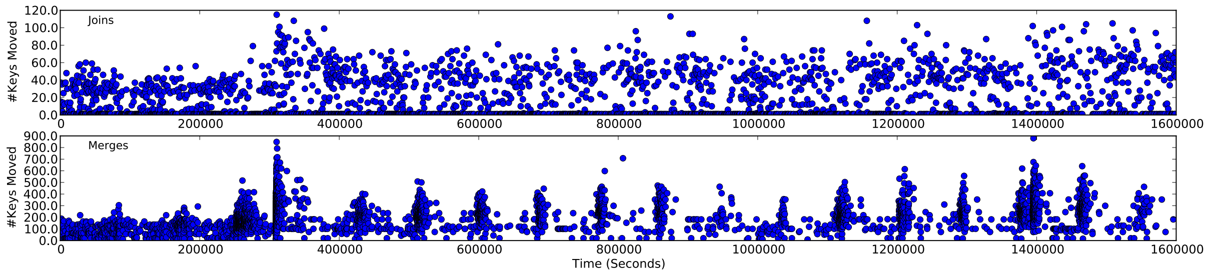
- Globally, it is clear that all group-based approaches have the common trend of creating merges when the network size decreases. In fact, the size of the network follows a diurnal-nocturnal pattern (there are more active nodes during the day), and during the transition to the night, the network shrinks and more merges are created. This is particularly clear around the $300,000$ seconds mark, where a strong descent on the number of nodes causes, not only an increase in the number of merges, but also on the number of keys moved by each merge.

- Regarding network joins, HonE shows consistently lower values for the number of keys moved than R-LB and Supersize-me, which means that, more frequently, when nodes join, they actually receive very few keys. Furthermore, since HonE makes an active effort to maintain the network relatively stable, it is also able to achieve fewer merges than R-LB (about 30% less on average), and most merges end up moving as many keys as in R-LB.

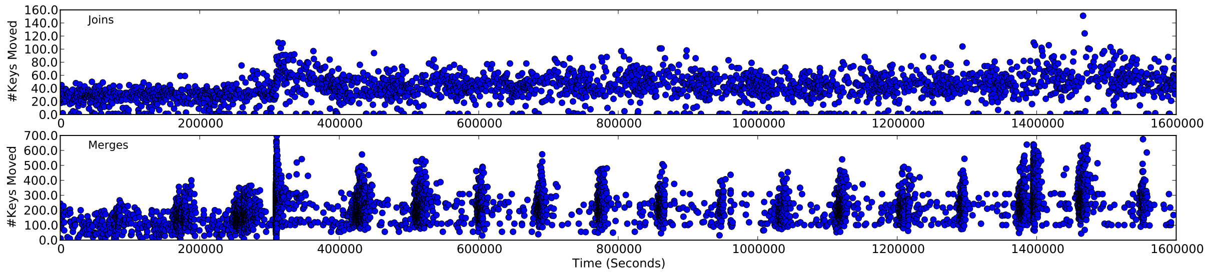
- Supersize-me leads to considerably fewer merges. In fact, except for situations when the network size drops abruptly, Supersize-me is able to mostly avoid them. However, it is also observable that the increased size of the groups has a negative effect on the number of keys moved as a consequence of joins.



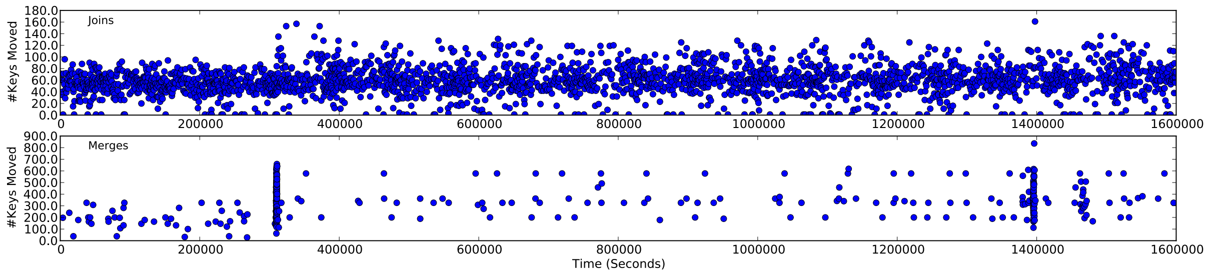
a) Network size over time.



b) Behaviour of HonE over time.



c) Behaviour of R-LB over time.



d) Behaviour of Supersize-me over time.

Figure 3.8: Behaviour of policies over time

3.10 Summary

This chapter proposed a novel combination of gossip-based mechanisms and structured overlays to generate a DHT of virtual nodes, where each virtual node is materialized by a set of physical nodes. The resulting system can achieve greater placement flexibility than previous solutions, not at the expense at fault-tolerance. Our solution, named Rollerchain, was experimentally proved to be more robust than competing approaches.

Using this new architecture, we studied different replication policies and provided interesting insights on the tradeoffs involved. Based on these insights, we were able to propose a novel policy, named "Hotter-On-Ephemeral", which significantly outperforms previous work.

Notes

The results for Rollerchain presented in this chapter were accomplished in cooperation with João Leitão. The preliminary design of the overlay was first presented as a poster, with the title "Rollerchain: a DHT for High Availability", in the 11th ACM/IFIP/Usenix Middleware Conference, Lisbon, Portugal, December 2010. The final protocol and its evaluation was proposed in the paper "Rollerchain: a DHT for Efficient Replication", Proceedings of the 12th IEEE International Symposium on Network Computing and Applications, Cambridge, MA USA, August 2013.

The results for the replication policies presented in this chapter were and proposed in the paper "Policies for Efficient Data Replication in P2P Systems", Proceedings of the 19th IEEE International Conference on Parallel and Distributed Systems, Seoul, Korea, December 2013.

4 Datacenter Scale Data Placement

The datacenter scale scenario is characterized by having thousands of nodes connected by stable links on controlled network infrastructures. In this context, distributed NoSQL key-value stores (DeCandia et al., 2007; Lakshman and Malik, 2010; Cooper et al., 2008; Marchioni and Surtani, 2012) have emerged as the reference architecture for data management in the cloud. A data placement algorithm for such systems must simultaneously address two main, typically opposing, concerns: i) maximizing locality by replicating data at the nodes that access them more frequently, while enforcing constraints on the object replication degree and on the capacity of nodes; ii) maximizing lookup speed, by ensuring that a copy of an object can be located as quickly as possible.

The data placement problem has been investigated in a number of alternative variants, e.g. Dowdy and Foster, 1982, Krishnan et al., 2000. Classic approaches formulate the data placement problem as a constraint optimization problem, and use Integer Linear Programming techniques to identify the optimal placement strategy with the granularity of single data items. Even though the datacenter scale scenario allows for solutions with more placement flexibility than the internet scale, these approaches suffer from several practical limitations. In the first place, finding the optimal placement is a NP-hard problem, hence any approach that attempts to optimize the placement of each and every item is inherently non-scalable. Further, even if the optimal placement could be computed, it is challenging to efficiently maintain a (potentially very large) directory to store the mapping between items and storage nodes.

Directories are indeed used by several systems such as PNUTS (Cooper et al., 2008) or BigTable (Chang et al., 2008). To minimize the costs associated with directory maintenance, these systems trade-off placement flexibility and support placement at a very coarse level, i.e., large data partitions rather than on a per-instance basis. However, even if coarse granularity is used, the use of a directory service introduces additional round-trip delays along the critical execution path of data access operations, which can considerably hinder performance.

To avoid the above issues, many popular key-value stores, such as Cassandra (Lakshman and Malik, 2010), Dynamo (DeCandia et al., 2007), Infinispan (Marchioni and Surtani, 2012), use random placement based on consistent hashing. By relying on random hash functions to determine the location of data across nodes, these solutions allow lookups to be performed locally, in a very efficient manner (DeCandia et al., 2007). However, due to the random nature of data placement (oblivious to the access frequencies of nodes to data), solutions based on consistent hashing may result in highly sub-optimal data placements.

This chapter presents AutoPlacer, a system aimed at self-tuning data placement in a distributed key-value store, which introduces a set of novel techniques to address the trade-offs described in the previous paragraphs. AutoPlacer employs a lightweight distributed optimization algorithm. The algorithm operates in rounds, and, in each round, it optimizes, in a decentralized fashion, the placement of the top-k "hotspots", i.e., the objects generating most remote requests, for each node of the system. In order to be able to identify the "hotspots" of each node with low processing cost, AutoPlacer adopts a state-of-the-art stream analysis algorithm (Metwally et al., 2005) that permits to track the top-k most frequent items of a stream in an approximate, but efficient manner. The information provided by the Space-Saving Top-k algorithm is then used to instantiate the data placement optimization problem.

Unlike solutions that rely on directory services, AutoPlacer guarantees 1-hop routing latency. To this end, AutoPlacer combines the usage of consistent hashing, which is used as the default placement strategy for less popular items, with a highly efficient, probabilistic mapping strategy that operates at the granularity of the single data item, achieving high flexibility in the relocation of (a possibly very large number of) hotspot items.

The key innovative solution introduced to pursue this goal is a novel data structure, which we named *Probabilistic Associative Array* (PAA). The goal of the PAA is to minimize the cost of maintaining a mapping associating keys with nodes in the system. PAAs expose the same interface of conventional associative arrays, but, in order to achieve space efficiency, they trade-off accuracy and rely on probabilistic techniques which can lead to inaccurate results with a user-tunable probability (these inaccuracies do not affect the accuracy of the key lookups in AutoPlacer: in the worst case they may only degrade the system's performance).

In summary, AutoPlacer provides two key features:

- It introduces a novel iterative, decentralized, self-tuning data placement optimization scheme;
- It preserves efficient lookups, while achieving high flexibility in determining an optimized data placement, through the use of a new probabilistic data structure designed specifically for this purpose.

AutoPlacer has been integrated in a popular, open-source key-value store, namely Infinispan: RedHat Infinispan is the reference NoSQL platform and clustering technology for JBoss AS, a mainstream open source J2EE application server. The choice of Infinispan is motivated by insights we got from use cases, internals, and limitations of this system as a result of a joint project with RedHat (Romano et al., 2014). We conducted experiments on both public and private cloud infrastructures, using a porting of the well-known TPC-C benchmark for key-value stores (Leutenegger and Dias, 1993), and GeoGraph (Ziparo et al., 2013), a complex benchmark representative of Geo-social network applications. The results of our experimental study highlight the effectiveness of AutoPlacer, which can achieve up to 6x, speed-ups with respect to a baseline system using consistent hashing.

The rest of the chapter is structured as follows. We start by placing the related work into context with AutoPlacer in Section 4.1. Our target system is characterized in Section 4.2. Section 4.3 provides a global overview of AutoPlacer. Then, its components are described in more detail in the next two sections: the PAA internals are described in 4.4; a theoretical analysis of the optimizer's accuracy is provided in 4.5. Section 4.6 reports the results of the experimental evaluation of the system. Finally, Section 4.7 concludes the chapter.

4.1 Context

This section puts the related work into context in relation to AutoPlacer. It focuses on how the state-of-the-art systems support dynamic changes in data placement and how these are used to improve the locality of data access.

As previously mentioned, a common approach to implementing data placement mechanisms in datacenter-scale systems is to rely on coarse-grained, user-defined data partitions/buckets, also called directories (Corbett et al., 2012) or tablets (Cooper et al., 2008; Chang et al., 2008). This mechanism is mostly used to balance the load on hotspot nodes through centralized control. While coarse partitioning allows for somewhat manageable directories, its coarse granularity can reduce the effectiveness of the load balancing mechanisms. In order to improve data locality, these systems also make use of sorted keys: the programmer is responsible for assigning similar keys to related data in order for it to be placed in the same server (or in the same group of servers) (Corbett et al., 2012; Cooper et al., 2008; Lakshman and Malik, 2010). The proposed system does not require the programmer to manually define data partitions. While AutoPlacer exploits information encoded in the key structure, the programmer is not required to define partitioning rules. Conversely, these are automatically inferred using machine learning techniques based on the data access patterns exhibited by the various nodes in the system (more details can be found on Section 4.4.2). Also, by inferring data partitioning rules using an online decision tree algorithm, our system can establish a fine grained placement for the most accessed items in a space-efficient way.

There is extensive work on defining optimal data placement strategies in multiple contexts. A naive design that achieves an optimal object-to-node mapping is to rely on a centralized component. In such a scheme, nodes send their object request to a single node, which runs an optimization algorithm based on some variant of the File Allocation Problem (FAP) (Chandy and Hewes, 1976; Leff et al., 1993; Zaman and Grosu, 2011; Laoutaris et al., 2006), and then the new mappings are propagated to a router component which creates lookup tables (Tatarowicz et al., 2012) and routes requests to system nodes. However, not only does this solution involve an extra network hop for each request, but also FAP is an NP-complete problem (hence, inherently non-scalable in terms of the number of objects) and this design would cause nodes to send information on every single object they store to a centralized location. Finally, Leff et al., 1993 show that such an algorithm can react slowly to changes in workloads since the decisions are not local to the nodes. Despite these drawbacks, several state-of-the art works follow approaches inspired on this naive solution: Jia et al., 2013 and Cruz et al., 2013 use a centralized component which collects fine-grained statistics on object usage, derives a placement using an optimization algorithm for improving load balancing across the system and then acts as a directory to locate objects. Ursa (You et al., 2013) also follows a similar design for achieving load balancing goals, but it only considers the most-used objects in order to improve scalability, in a similar way as AutoPlacer uses top-k. Schism (Curino et al., 2010) and the work by Pavlo et al., 2012 also follow this centralized design, with a similar goal as AutoPlacer of improving data locality. To achieve optimal placement, these latter two systems collect system execution logs to extract statistics on correlated objects, and then use graph partitioning algorithms to place the objects into groups which are then mapped to nodes. Despite achieving near-optimal placements, these are mostly offline algorithms, which consider all data in the system and have limited scalability as the number of data items grows.

Recent work by Li et al., 2013 proposes Proteus, a virtual node placement algorithm that allows minimizing the delay penalties otherwise incurred during server provisioning dynamics when using a conventional consistent hashing placement scheme. This scheme could be easily integrated with AutoPlacer, as a mechanism like Proteus could be used in AutoPlacer as alternative to the consistent hashing scheme.

Finally, this work is also related to the vast literature on self-management (Cook et al., 2012; Forell et al., 2011) of cloud data platforms, and in particular to the problems of thermal optimization (Li et al., 2011; Chen et al., 2011), and SLA-based provisioning (Wang et al., 2011; Didona et al., 2012). The self-tuning mechanisms designed to cope with these problems can induce significant variations of the locality patterns exhibited by the nodes of a distributed key-value store, e.g., by dynamically altering the amount of allocated resources, or the way in which requests are dispatched to resources. AutoPlacer can be used to ensure that, whenever these systems reconfigure the platform to meet SLA or thermal constraints, the placement of data across the nodes of the system is always constantly optimized.

4.2 System Characterization

The development of AutoPlacer has been motivated by our experience (Peluso et al., 2012b; Peluso et al., 2012a; Ruivo et al., 2011) with the use of an existing, state-of-the-art, key-value store, namely Infinispan (Marchioni and Surtani, 2012) by Red Hat. In Infinispan (and other similar products such as (Lakshman and Malik, 2010; DeCandia et al., 2007)), data is stored in multiple nodes using consistent hashing. For each key, consistent hashing determines a *supervisor* node for that item. Items can be replicated. A node that stores a copy of data item i is denoted an *owner* of that item. Assume that d copies are maintained of each data item, the owners of data item i are deterministically assigned to be j's supervisor plus its $d-1$ immediate successors (in the one-hop distributed hash-table that is used to implement consistent hashing).

Each node serves a dual purpose: it stores a subset of the data items maintained by the distributed store and also executes application code. The application code may be structured as a sequence of *transactions* (Infinispan supports transactional properties), with different isolation levels.

When the application code reads a data item, its value must be retrieved from one of its owners (which can be another node in the cluster). Thus, optimal performance is achieved if the node that executes a given application is the owner for the items it accesses more often. When the application writes a data item, all owners must be updated. Interestingly, the placement policy can also affect the performance of write operations. When multiple writes are performed in the context of a transaction, they can be applied in batch when the transaction commits. Hence, the larger the number of owners of keys updated by a transaction, the higher the number of nodes that have to be contacted during its commit phase.

Infinispan uses consistent hashing to ensure that all lookups can be executed locally. Unfortunately, in typical deployments of large-scale key-value stores, random data placement can be largely suboptimal as applications are likely to generate skewed access distributions (Leutenegger and Dias, 1993), often dependent on the actual "type" of operations processed by each node (You et al., 2013; Curino et al., 2010). Also, workloads are frequently distributed according to load balancing strategies that strive to maximize locality (Garbatov and Cachopo, 2011)/minimize contention (Amza et al., 2003). As we will show in the evaluation section, all these facts make consistent hashing sub-optimal. Therefore, significant performance improvements can be achieved by using appropriate autonomic data placement strategies.

4.3 AutoPlacer Overview

AutoPlacer is designed to optimize data location in a decentralized manner, i.e., each node in the system contributes to the global optimization process. Since AutoPlacer is aimed at systems that use consistent hashing as the default data placement policy, we also rely on consistent hashing to decentralize the optimization effort: each node is responsible for deciding the placement for the items it supervises. AutoPlacer executes, periodically, a sequence of optimization rounds. As a result of each round, a number of data items may be relocated. This happens only if the expected gains are above a minimum threshold. Each optimization round consists of the following sequence of six tasks.

Task 1: The first task of the AutoPlacer approach consists of collecting statistics about the *hotspots* data, i.e., the top-k most accessed data items, at each node. In fact, instead of trying to optimize the placement of every data item in a single round, at each optimization round, AutoPlacer only optimizes the placement of items that are identified as hotspots. Since this task is run periodically, once some hotspots have been identified (and relocated) in a given round, new (different) hotspots are sought in the next round. Therefore, although in each round only a limited number of hotspots is identified, in the long run, a large number of data items may be selected over multiple optimization rounds, as long as gains can still be obtained from their relocation.

Task 2: The second task consists of having the nodes exchange statistics regarding the data items that were identified as hotspots during the current round. More precisely, each node gathers (from the remaining nodes of the platform) access statistics on any hotspot items it supervises.

Task 3: The above information is used in the third task (denoted the *optimization* task) to find an appropriate placement for those items. The result of this task is a *partial relocation map*, i.e., a mapping of where replicas of each hotspot items that the node supervises (for the current round) must be placed.

Task 4: Even if the number of hotspots tracked at each round is a small fraction of the entire set of items maintained in the key-value store, over multiple rounds the relocation map can grow in an undesired way, and may even be too large to be efficiently distributed to all nodes. This task is devoted to encoding the relocation map in a probabilistic data structure that can be efficiently replicated on all nodes in order to ensure fast lookups, i.e., a *Probabilistic Associative Array* (PAA). Specifically, each node computes the PAA for the (relocated) objects it supervises.

Task 5: Once each PAA has been computed, each node disseminates it among all nodes. By assembling the PAAs received from all the nodes in the system, each node can locally build an object lookup table that includes updated information on the placement of data optimized during this round.

Task 6: Finally, at the end of each round, the data items for which new locations have been derived are transferred (using conventional state-transfer facilities (Jiménez-Peris et al., 2002; Ahmad et al., 2013)) in order to match the new data placement.

As can be inferred from the previous description, the work is divided among all nodes and communication takes place only during tasks 2, 5, and 6, in order to, respectively, exchange statistical information on hotspots, distribute the PAA, and finally relocate the objects. Also, the tasks that require communication are performed in parallel, without the help of any centralized component.

Note that AutoPlacer is only concerned with the placement of data and is agnostic to the data consistency mechanisms used by the key-value store. For instance, AutoPlacer is compatible with the different consistency levels supported in Infinispan. Naturally, the consistency protocol in use must be prepared to support dynamic re-adjustment of the number and placement of replicas, but this is today the default for most consistency protocols, as it is required anyway for fault-tolerance.

In the next subsections, we provide more information about the two main components of AutoPlacer, namely, the optimizer (executed by Task 3) and the PAA (built in Task 4 and used subsequently to perform data lookups locally).

4.3.1 Optimizer

Most works, e.g., You et al., 2013, Leff et al., 1993, Krishnan et al., 2000, Dowdy and Foster, 1982, in the area of data placement (and of its many variants (Krishnan et al., 2000; Dowdy and Foster, 1982)) assume that the objective and constraint functions of the optimization problem can be expressed (or approximated) via linear functions, and, accordingly, formulate an Integer Linear Programming (ILP) problem. The ILP model can indeed be also adopted for the specific data placement problem tackled by AutoPlacer. To this end, one can model the assignment of data to nodes by means of a binary matrix X, in which $X_{ij} = 1$ if the object i is assigned to node j, and $X_{ij} = 0$ otherwise. Further, one can associate (average, or per-object) costs with local/remote read/write operations. For simplicity of exposition, we consider only objects with fixed size. The ILP problem is then formulated as the minimization of the objective function that expresses the total cost of accessing all data items across all nodes, subject to two constraints:

Table 4.1: Parameters used in the ILP formulation.

$\mathcal{N}$	the set of nodes j in the system
$\mathcal{D}$	the set of objects i in the system (Data-Set)
X	a binary matrix in which $X_{ij} = 1$ if the object i is assigned to node j, and $X_{ij} = 0$ otherwise
r_{ij}, w_{ij}	the number of read, resp. write, accesses performed on an object i by node j
cr^r, cr^w	the cost of a remote read, resp. write, access
cl^r, cl^w	the cost of a local read, resp. write, access
d	the replication degree, that is, number of replicas of each object in the system
S_j	the capacity of node j.

i) the number of replicas of each object must meet a predetermined replication degree, and ii) each node has a finite capacity (it must not be assigned more objects than it can store). In Table 4.1, we list the parameters used in the problem formulation, which aims at minimizing the following cost function:

$$\sum_{j\in\mathcal{N}}\sum_{i\in\mathcal{D}} \overline{X}_{ij}(cr^r r_{ij} + cr^w w_{ij}) + X_{ij}(cl^r r_{ij} + cl^w w_{ij}) \tag{4.1}$$

subject to:

$$\forall i \in \mathcal{D} : \sum_{j\in\mathcal{N}} X_{ij} = d \wedge \forall j \in \mathcal{N} : \sum_{i\in\mathcal{D}} X_{ij} \leq S_j$$

Despite its convenient mathematical formulation, ILP problems are NP-hard. Further, solving the above ILP problem would require collecting and exchanging among nodes access statistics for all objects in the system. We tackle these drawbacks by introducing a lightweight, multi-round distributed optimization algorithm, which we describe in the following Section 4.3.1.1.

4.3.1.1 Space-Saving Top-k algorithm

An important building block of AutoPlacer is the Space-Saving Top-k algorithm by Metwally et al., 2005. This algorithm is designed to estimate the access frequencies of the top-k most popular objects in an approximate, but very efficient way, i.e., by avoiding maintaining information on the access frequencies (namely counters) for each object in the stream. Conversely, the Space-Saving Top-k algorithm maintains a tunable, constant, number m, where $m \ll |\mathcal{D}|$, of counters, which makes it extremely lightweight. On the downside, the information returned in the top-k list may be inaccurate in terms of both the elements that compose it and their estimated frequency. However, this algorithm has a number of interesting properties concerning the inaccuracies it introduces. First, it ensures that the access frequencies of the objects it tracks are always consistently overestimated. Also, its maximum overestimation error is known, and is equal to the frequency of the least frequently accessed item present in top-k, denoted as F_k. Finally, its space-requirements can be tuned to bound the maximum error introduced in the frequency tracking, as we will further discuss in Section 4.5.

4.3.1.2 Using Approximate Information

In AutoPlacer, each node j runs 2 distinct instances, noted as $\textit{top-}k_j^{rd}$, resp. $\textit{top-}k_j^{wr}$, of the Space-Saving Top-k algorithm, used to track the k most frequently read, resp. updated, data items during the current optimization round. We denote with $\textit{top-}k_j(\mathcal{D})$ the subset of cardinality k (of the entire data set $\mathcal{D}$) contained in both the read and write top-k instances at node j, and with $\textit{top-}K(\mathcal{D}) = \cup_{j\in\mathcal{N}}(\textit{top-}k_j(\mathcal{D}))$ the union of the top-k data items across all nodes.

By restricting the optimization problem to the top-k accessed data items we reduce the number of decision variables of the ILP problem significantly, namely from $|\mathcal{D}||\mathcal{N}|$ to $O(k|\mathcal{N}|)$ (where $k \ll |\mathcal{D}|$). This choice is crucial to guarantee the scalability of the proposed approach. However, it requires dealing with the incomplete and approximate nature of the data (read/write) access statistics provided by the top-k algorithm, which we denote with $\hat{r}_{ik}$,$\hat{w}_{ik}$ to distinguish them from their exact counterparts (r_{ik},w_{ik}). Also, we use the notation $\hat{X}$ to refer to the solution of the optimization problem using as input the access statistics provided by the top-k algorithm, and distinguish it from the one obtained using the exact access statistics in input, which we denote X^{opt}.

A first problem to address is related to the possibility of missing information concerning the access frequency by some node j to some data item $i \in \textit{top-}K(\mathcal{D})$: this can happen in case i has not been tracked in $\textit{top-}k_j(\mathcal{D})$, but is present in the $\textit{top-}k_{j'}(\mathcal{D})$ of some other node $j' \neq j$. To address this issue, we simply set to 0 the frequencies $\hat{r}_{ij}$,$\hat{w}_{ij}$.

Finally, the approximate nature of the information provided by the Space-Saving Top-k algorithm may impact the quality of the identified solution. A theoretical analysis aimed at evaluating this aspect will be provided in Section 4.5.

4.3.1.3 Accelerating the solution of the optimization problem

To accelerate the solution of the optimization problem we take two complementary approaches: relaxing the ILP problem, and parallelizing its solution.

The ILP problem requires decision variables to be integers and is computationally onerous (You et al., 2013). Therefore, we transform it into an efficiently solvable linear programming (LP) problem. To this end, we let the matrix $\hat{X}$ assume real values between 0 and 1 (adding an extra constraint $\forall i \in \mathcal{D}, \forall j \in \mathcal{N}\ 0 \leq \hat{X}_{ij} \leq 1$). Note that the solutions of the LP problem can have real values, hence each object is assigned to the d nodes which have highest $\hat{X}_{ij}$ values. As in the work by You et al., 2013, we use a greedy strategy according to which, if the assignment to a node causes a violation of its capacity constraint, the assignment is iteratively attempted to the node that has the $(d+k)$-th ($k \in [1, |\mathcal{N}| - d]$) highest scores.

Second, we introduce a controlled relaxation of the capacity constraint, which allows us to partition the ILP problem into $|\mathcal{N}|$ independent optimizations problems that we solve in parallel across the nodes of the platform. Let $\textit{top-}k_j(\mathcal{D}|n)$ be the set of keys in $\textit{top-}k_j(\mathcal{D})$ of node j that node n supervises. Each node j sends its $\textit{top-}k_j(\mathcal{D}|n)$ to each other node n in the system. As a result, each node j also gathers the access statistics $\textit{top-}K(\mathcal{D}|j) = \cup_{n\in\mathcal{N}}\ \textit{top-}k_n(\mathcal{D}|j)$ concerning the current hotspots that j supervises. At this point, each node j computes the new placement for the data in $\textit{top-}K(\mathcal{D}|j)$.

Note that since we are instantiating the (I)LP optimization problems in parallel, and in an independent fashion, we need to take an additional measure to guarantee that the capacity constraints are not violated. To this end, we instantiate the (I)LP problems at each node j with a capacity $S'_j = S_j - |\mathcal{N}|k$. In practice, this relaxation is expected to have minimum impact on the solution quality, since $k \ll S_j$.

Overall, at the end of an optimization round, each node j produces two outputs: the partial relocation map $\hat{X}$, and the cost reduction achievable by relocating the data in $\textit{top-}K(\mathcal{D}|j)$ according to $\hat{X}$, which we denote as Δ_{C^j}. Δ_{C^j} is computed as the difference between the result of Equation 4.1 applied to the partial relocation map $\hat{X}$ obtained in this round, and the map obtained in the previous round. This value allows estimating the gain achievable by performing this optimization round, and is used in AutoPlacer to determine the completion of the round-based optimization algorithm.

Table 4.2: PAA Interface.

Method	**Input Parameters**	**Output**
CREATE	Set⟨Key,Value[d]⟩, α, β	PAA
GET	Key	Value[d]
ADD	Set⟨Key,Value[d]⟩	PAA
GETDELTA	PAA	ΔPAA
APPLYDELTA	ΔPAA	PAA

4.3.2 Probabilistic Associative Array: Abstract Data Type Specification

Even though in each round AUTOPLACER optimizes the placement of a relatively small number of data items, over multiple optimization rounds the number of relocated objects can grow very large. Hence, a relevant issue is related to the overhead for maintaining, and replicating, a possibly very large relocation map. Indeed, the relocation map can be seen as an associative array in which each entry is a pair mapping a data item to the set of nodes that own it.

The Probabilistic Associative Array (PAA) is a novel data structure that allows maintaining an associative array in a space efficient, but approximate way. We present the PAA as an abstract data type, with an interface analogous to conventional associative arrays. In Section 4.4, we will discuss how it has been implemented in AUTOPLACER.

The PAA is characterized by the API reported in Table 4.2, which is similar to that of conventional associative arrays, including methods to create and query a map between keys and (constant d-sized) arrays of values. To this end, the PAA API includes three main methods:

- the CREATE method, which returns a new PAA instance and takes as input a set of pairs in the domain (key $\times$ array[d] of values) to be stored in the PAA (called, succinctly, *seed map*) and two tunable error parameters α and β (discussed below);
- the GET method, which allows querying the PAA obtaining the array of values associated with the key provided as input, or $\perp$ if the key is not contained in the PAA;
- the ADD method, which takes an input a seed map and adds it to an existing PAA.

The PAA is designed to tradeoff accuracy for space efficiency, and may return inaccurate results when queried. In the following, we specify the properties ensured by the GET method of a PAA:

- it may provide *false positives*, i.e., it can provide a return value different from $\perp$ for a key that was not inserted in the PAA. The probability of false positives occurring is controlled by parameter α.
- it has no *false negatives*, i.e., it will never return $\perp$ for a key contained in the seed map.
- it may return an *inaccurate* array of values for a key contained in the seed map. The probability of returning inaccurate arrays is controlled by parameter β. In other words, with some small and tunable probability, the data items may be located in different nodes than those specified by the seed map (thus, the lookup efficiency may cause some degree of sub-optimal placement).
- its response is deterministic, i.e., for a given instance of a PAA, the return value for any given key is always the same.

Finally, the PAA API contains two additional methods that allow to update the content of a PAA in an incremental fashion: GETDELTA, and APPLYDELTA. GETDELTA takes as input a PAA and returns an encoding, denoted as ΔPAA, of the differences between the base PAA over which the method is invoked (say PAA$_1$) with respect to the input PAA, say PAA$_2$. The ΔPAA returned by GETDELTA can then be used to obtain PAA$_2$ by invoking the method APPLYDELTA over PAA$_1$ and passing as input parameter ΔPAA.

Before presenting the internals of the PAA implementation (see Section 4.4), we discuss, in the next sections, the functioning of AUTOPLACER's distributed algorithm.

4.3.3 The AutoPlacer iterative algorithm

We now provide, in Algorithm 1, the pseudo-code formalizing the behavior of the AUTOPLACER algorithm executed by a node j. Each node maintains a local lookup table, denoted as *LookupT*, that consists of an array of PAAs, one per each node j in the set of nodes composing the system (denoted by Π in Alg. 1). Specifically, j's entry of *LookupT* is used to keep track of the objects supervised by node j that have already been relocated by AUTOPLACER. For any given round, *LookupT* is the same on all nodes.

At the beginning of each round, j collects statistics concerning its top-k most frequently read/written data items. This activity is encapsulated by the `collectStats` procedure (line 5), which is designed to track only accesses to objects whose placement had not been previously optimized in previous rounds. This measure is necessary, as, otherwise, in the presence of stable distributions of the data access among nodes (i.e., stable workloads), the top-k lists at each node may quickly stagnate. Especially in case of skewed distributions, the top-k lists would tend to track the very same objects (i.e., the most popular ones) along every round.

By tracking only the keys whose placement has not been optimized in previous rounds, it is guaranteed that, in two different rounds, two disjoint sets of objects are considered by the optimization algorithm, leading to the analysis of progressively less "hot" data items. Further, it prevents the possibility of ping-pong phenomena (Fleisch and Popek, 1989), i.e., the continuous re-location of a key between nodes, as it guarantees that the position of each object is optimized at most once.

To determine whether an access to a data item should be traced or not, the `collectStats` procedure is provided with *LookupT* as input (we recall that *LookupT* keeps track of all items whose placement has been previously optimized). Upon a read/write access to a data item, the `collectStats` procedure checks if the item is contained in *LookupT* and, in the positive case, it avoids tracing this access. Notice that this assumes that the data access frequencies do not change significantly during the entire optimization process. In fact, in order to cope with scenarios in which applications' data access patterns change at a frequency higher than AutoPlacer's complete optimization, the structure of *LookupT* must be more complex. In Section 4.3.4, we detail how these scenarios are handled by AutoPlacer.

Next, the nodes exchange the information collected by `collectStats`. Since we also parallelize the optimization procedure, we send to each node only the statistical information that will be relevant to the computation that will be performed at that node, i.e., the statistical information regarding the data items it supervises.

At this point, each node optimizes the placement for the objects it supervises (`Optimize` primitive), determining their new owners (encoded in the partial relocation map, denoted $\hat{X}$). The node also computes the reduction of the local cost function (denoted as Δ_{C^j}) that the new assignment brings.

Then, node j computes a temporary PAA, based on the previous value of its PAA (stored in $LookupT[j]$) and on the new additional relocation information $\hat{X}$ (lines 13-14). The API of the PAA is then used to extract the relevant deltas from the existing PAA that need to

ALGORITHM 1: AutoPlacer's behavior at node j

```
Array[1...|$\mathcal{N}$|] of PAA : LookupT={$\perp, \ldots, \perp$};
PAA: tmpPAA=$\perp$;
do
    Array[1...|$\mathcal{N}$|] of Set$\langle i, r, w \rangle$ : req=$\perp$;
    $\langle top_k^{rd}, top_k^{wr} \rangle \leftarrow$ collectStats(LookupT);
    foreach $n \in \Pi$ do
        send({$\langle i, r, w \rangle \in \{top_k^{rd} \cup top_k^{wr}\}$ such that supervisor($i$) = $n$}) to $n$;
    end
    foreach $n \in \Pi$ do
        req[j]$\leftarrow$ receive() from $n$;
    end
    $\langle \hat{X}, \Delta_{C^j} \rangle \leftarrow$ Optimize(req);
    tmpPAA $\leftarrow$ LookupT[$j$];
    tmpPAA.ADD($\hat{X}$);
    $\Delta$PAA: delta $\leftarrow$ tmpPAA.GETDELTA(LookupT[$j$]);
    broadcast(delta,$\Delta_{C^j}$);
    $\Delta_{C^*} \leftarrow 0$;
    foreach $n \in \Pi$ do
        [delta,$\Delta_{C^n}$] $\leftarrow$ receive() from $n$;
        LookupT[$n$]$\leftarrow$LookupT[$n$].APPLYDELTA(delta);
        $\Delta_{C^*} \leftarrow \Delta_{C^*} + \Delta_{C^n}$;
    end
    moveData();
while $\Delta_{C^*} > \gamma$;
```

be disseminated, in order to avoid sending the entire PAA again (line 15). These deltas are exchanged among nodes, and applied locally, such that every node can update all entries of *LookupT* (lines 16-22).

Each optimization round ends by triggering the re-location of the data via the `moveData`() primitive. This primitive will use the updated PAAs to determine the set of items that have been re-located, and gives the necessary commands to perform the corresponding state transfers. Several state transfer techniques can be used for this purpose (Jiménez-Peris et al., 2002; Ahmad et al., 2013), whose complexity depends on the consistency guarantees that the key-value store implements (e.g. transactional vs eventual consistency). These mechanisms are orthogonal to the AutoPlacer system. In the Infinispan version used for evaluation of the current work, the state transfer is achieved through an algorithm which temporarily buffers the affected operations during the state transfer phase and re-routes them to the correct nodes after the state transfer, hence causing no consistency losses. Techniques to optimize this procedure may be found in the related work (Ahmad et al., 2013; Jiménez-Peris et al., 2002; RedHat/JBoss, 2013).

ALGORITHM 2: PAA-based lookup function

```
Array[1 ... $d$] of Nodes LOOKUP(Key $k$)
    if LookupT[supervisor($k$)].GET($k$) $\neq \perp$ then
        return LookupT[supervisor($k$)].GET($k$);
    else
        s $\leftarrow$ supervisor($k$);
        return {s, s+1, ..., s+d-1};
    end
end
```

AutoPlacer relies on a mechanism that halts the distributed optimization algorithm if the "gain" achieved during the last optimization round does not exceed a user-tunable minimum threshold, denoted γ (line 23). This allows avoiding analyzing the "tail" of the data access distribution, whose optimization would lead to negligible gains. We chose as metric to evaluate the optimization gain the reduction of the cost function achieved during the last optimization round, Δ_{C^*}. To compute this value, each node j disseminates the value for the reduction of its local cost function Δ_{C^j} along with *delta* (line 16). After this dissemination phase, each node can deterministically compute Δ_{C^*} and evaluate the predicate on the termination of the optimization algorithm.

Finally, AutoPlacer leverages the fault-tolerant mechanisms used by the underlying key-value store itself to recover from node failures. In fact, Infinispan relies on JGroups (Ban and Blagojevic, 2002) to maintain cluster membership; if a node crashes, both the key-value store and the AutoPlacer components receive consistent up-to-date membership change notifications that can be used to safely recover from failures (for instance, this prevents AutoPlacer from blocking while waiting for inputs from different nodes).

After one round of the AutoPlacer algorithm, some items will be mapped to nodes different from those mapped by consistent hashing. Hence, the system must use a lookup function which supports this change. Algorithm 2 shows the pseudocode for the lookup function for a key k. First, consistent hashing is used to identify the supervisor of k, s. We then check whether the PAA associated with s contains k. In the positive case, we use the mapping information provided by *LookupT*[s] to identify the set of nodes that are currently maintaining key k. Otherwise, we simply return the set of owners for k as determined by consistent hashing (d is the replication degree).

4.3.4 Handling dynamic workloads

In the previous sections, we have described how AutoPlacer can be employed to optimize data placement in the presence of static workloads. However, the algorithm can be extended to cope with dynamic workloads, i.e., scenarios in which the data access patterns generated by the nodes in the system vary over time. In this case, the data placement identified by AutoPlacer at the end of a round may become suboptimal and require the re-location of data items whose placement had previously been optimized by AutoPlacer. The key issue is that, once a data item has been relocated by AutoPlacer, its data accesses are no longer traced via the top-k, in order to avoid re-optimizing its placement in following rounds and to prevent the stagnation of the top-k statistics. Next, we show how it is possible to extend AutoPlacer to cope with dynamic workloads, by having it operate in *epochs*.

In each epoch, AutoPlacer operates exactly as previously described. A new epoch is started when the need for recomputing data placement is identified, i.e., whenever an abrupt change of the access patterns is detected during the current epoch. Various key performance indicators may be adopted to reveal the occurrence of a relevant workload shift, including the ratio of remote to local data access operations and/or the number of nodes involved in commit. Indeed, as a consequence of AutoPlacer's optimization algorithm, these performance indicators are expected to decrease over time if the data access patterns at the various nodes remain stable.

In AutoPlacer, we trigger new epochs whenever we detect a sudden growth of the probability of remote data accesses. To this end, one could use various statistical mechanisms for robust change detection, such as the work by Sangyeol and Taewook, 2004. However, we experimentally found that using a simple threshold of a three-time increase in remote data accesses was sufficient to detect workload changes in a reliable way. Each node disseminates the remote data access information periodically in the system. At each dissemination round, every node computes the average remote access probability in the system, and uses the same deterministic function (i.e., comparison with the threshold) to determine whether to trigger a new epoch. Thus, the need to start a new epoch can be first detected by any node. In such case, the node sends a message to the remaining nodes to make sure all nodes will re-start executing Alg. 1. This simple scheme ensures that all nodes will agree on triggering a new epoch in the same dissemination round.

Upon beginning a new epoch e, each node creates an empty $LookupT^e$ associated with the current epoch. However, unlike a normal AutoPlacer round, this $LookupT^e$ is not merged with any pre-existing $LookupT$s, but it is stacked on top of $LookupT$ from previous rounds. This new $LookupT^e$ will serve a twofold purpose. On the one hand, it allows for storing a "patch" for the currently established data placement (encoded by the set of $LookupT$s of the preceding epochs). Furthermore, the current $LookupT^e$ is provided as input to the `collectStats()` method, which will use it to determine whether or not to track accesses to a data item d in the node's top-k in the current epoch, depending on whether d is stored in $LookupT^e$ (which, we recall, implies that d has been re-located by AutoPlacer during epoch e).

Since, in the first round of any epoch e, its $LookupT^e$ is empty, the accesses to *all* objects in the key-value store will be tracked, allowing to re-optimize the placement of data items that have been subjected to access patterns shifts since the last epoch. If objects need to be relocated, their new position is stored in the corresponding PAA in $LookupT^e$. At the end of the round, each node will broadcast the PAA associated with the data it supervises, as usual, and the other nodes will merge it with the empty $LookupT^e$. We also alter the way in which the lookup function operates: since an object may have been relocated in a previous epoch, we query the stack of $LookupT$ (in reverse chronological order). The lookup function returns the mapping determined by the first $LookupT$ that contains the required data item (i.e., the most recent re-location of a data item), or the result of the default consistent hashing scheme in case the data item has not been relocated in any epoch (and, therefore, is not present in any $LookupT$).

A simpler way for AutoPlacer to handle varying workloads would be to reset its $LookupT$ as it detects a change in the workload. Even though this would be an effective method of resetting AutoPlacer, it would have the negative effect of causing the system to shuffle all previously data optimized, returning to the original consistent hashing data placement. Keeping a history of previous $LookupT$, arranged in reverse chronological order and regulated by the notion of epochs, allows for minimizing the overhead introduced by the need to re-optimize the placement of data in the system. In fact, if the placement determined in a previous epoch remains optimal in the new epoch for a subset D of the platform's data, even though it is sub-optimal for a subset S, the data items in D will not be re-located in the new epoch (as the optimization algorithm executing during the new epoch will confirm the optimality of the current placement).

The disadvantage of keeping a history of $LookupT$ is that it could slow down the lookup function after a long series of epochs. However, it is important to note that after the system has stabilized on the most recent epoch e and AutoPlacer has stopped optimizing placement, $LookupT^e$ will contain the entire set of hotspots given the data access patterns currently exhibited by the application. Thus, PAAs belonging to previous epochs will only be queried to obtain the location of objects which were relocated in some previous epoch, and that are no longer used by the nodes they were moved to. Hence, after AutoPlacer has stabilized, all but the most recent $LookupT$ may be discarded, causing non-hotspot objects to return to their supervisor nodes (as determined by consistent hashing). Since the AutoPlacer algorithm is executed independently at each node, each node n may unilaterally decide when to purge the previous PAAs associated with the data it supervises. Thus, when a node has finished optimizing placement, it broadcasts a `PURGE` message so that other nodes will discard the existing history of PAAs associated with the objects n supervises.

Finally, we note that in order to ensure the termination of the AutoPlacer's optimization process, we need to assume that the frequency with which workloads change is sufficiently low to give AutoPlacer enough time to execute a sufficient number of optimization rounds. However, even in challenging scenarios characterized by frequent shifts of the application's data access patterns, the algorithm described above allows for effectively reacting to workload changes by minimizing the costs associated with the re-mapping of data replicas to nodes.

4.4 Probabilistic Associative Array Internals

4.4.1 Building Blocks

Scalable Bloom filters (SBF) (Almeida et al., 2007) are a variant of Bloom filters (BF) (Bloom, 1970), a well-know data structure that supports probabilistic test for membership of elements in sets. A BF never yields false negatives (if the query returns that an element was not inserted in a BF, this is guaranteed to be true). However, a BF may yield false positives (a query may return true for an element that was never inserted) with some tunable probability α, which is a function of the number of bits used to encode the BF and of the number of elements that one stores in it (that must be known a priori). SBFs extend BFs in that they can adapt their size dynamically to the number of elements effectively stored, while still ensuring a bounded false positive probability. Internally, this is achieved by creating, on demand, a sequence of BFs with increasing capacity.

VFDT (Domingos and Hulten, 2000) is a classifier algorithm that induces decision trees over a stream of data, i.e., without assuming the a priori availability of the entire training data set unlike most existing decision trees (Mitchell, 1997). VFDT is an incremental online algorithm, given that it has a model available at any time during its run and refines the model over time (by performing new splits, or pruning unpromising leaves) as it is presented with additional training data. As classical off-line decision trees, the output of VFDT is a set of rules that allows mapping a point in the feature space to a target discrete class. A noteworthy property of VFDT is that the trees it produces are asymptotically arbitrarily close to the ones produced by a batch learner (i.e., an off-line learner that uses the entire training set to determine how to grow the tree). The misclassification probability can be configured by means of the parameter δ (Domingos and Hulten, 2000), which affects the frequency with which new rules are induced in the trees built by the VFDT algorithm.

The PAA uses SBFs and VFDT in the following manner: SBFs are used to assert if a key was stored in the PAA; VFDT is used to obtain the set of values associated with a key stored in tha PAA. The following sections explain how this technique works in detail.

4.4.2 FeatureExtractor Key Interface

In order to maximize the effectiveness of the machine-learning statistical inference, programmers can optionally provide semantic information on the type of key inserted in a PAA, by having their keys implementing the FeatureExtractor interface. This interface exposes a single method, getFeatures(), which returns a set of pairs $\langle$*featureName*, *featureValue*$\rangle$, where *featureName* is a unique string identifying each feature and *featureValue* is a (continuous or discrete) value defining the value of that feature for the key.

The purpose of this interface is to allow a key to be mapped, in a semantically meaningful way (and hence inherently application dependant), into a multi-dimensional feature-space that can be more efficiently analyzed and partitioned by a statistical inference tool. In particular, the set of features extracted via this interface defines the "feature space" (Mitchell, 1997) over which, as we will see in the following section, the VFDT algorithm infers a set of compact rules. This ruleset, which is encoded as a decision tree, allows defining which regions of the feature space should be assigned to the various nodes of the system.

Normally, features can be "naturally" derived from the data model used in the application. For instance, if an object-oriented (or relational) model is used, a typical encoding for the key corresponding to an object of class "Person" with ID=3 may be "`Person-3`". The FEATURE-EXTRACTOR interface can then simply parse the key and return the pair $\langle$"Person"$, id\rangle$. This can be further illustrated considering the real example of the TPC-C benchmark, which we used in our evaluation. In this case, a "Customer" object with id c_1 would be associated with a feature, $\langle$"Customer", $c_1\rangle$. Further, since in TPC-C a customer is statically registered in a "Warehouse" object, c_1 would have a second feature $\langle$"Warehouse", $w_1\rangle$, being w_1 the identifier of the warehouse where c_1 is registered. Hence, a different customer c_2 registered with a warehouse w_2 would be associated with the features $\langle$"Customer", $c_2\rangle$ and $\langle$"Warehouse", $w_2\rangle$, while the object representing warehouse w_2 itself would be associated with the features $\langle$"Customer", N/A$\rangle$ and $\langle$"Warehouse", $w_2\rangle$.

Note that this sort of feature extraction could be automated, provided the availability of information on the mapping between the application's domain model, in terms, e.g., of entities and relationships, and the underlying key/value representation. This can be done using annotations or domain specific languages, analogously to what is done in Object-to-Relational Mapping (ORM) solutions (Bauer and King, 2006).

Finally, we note that this approach could be used, at least in principle, even in case of completely unstructured keys. In such a case, the feature extraction phase would map the keys onto a uni-dimensional feature space that coincides with the keys' domain. However, as we mentioned in Section 4.4.1, the PAAs use VFDT, an online decision-tree based classifier algorithm, to infer a set of rules that map regions of the feature space onto nodes in the system. As in any machine learning problem (Bishop, 2006), the ability of the classifier to infer a compact, yet accurate ruleset is, in practice, strongly dependant on the quality of the used features, and on their capability to promote the identification of clusters. Although, generally speaking, there is no universal rule that determines how many features should be used to achieve optimal performance (Liu and Motoda, 1998), one may argue that, in most classification problems, using a uni-dimensional feature space is likely to limit the inference capabilities of classifier algorithms. This is the reason why AUTOPLACER's programming model promotes, via the FEATUREEXTRACTOR interface, the encapsulation of additional semantic information in the key structure.

4.4.3 PAA Operations

We can finally discuss how each of the methods specified by the PAA abstract interface introduced in Section 4.3.2 is implemented:

• CREATE: to instantiate a new PAA from a seed map, an SBF is first created, sizing it to ensure the target error rate α and populating it with the keys passed as input. Next, d new instances of VFDT are trained. The i-th instance of VFDT ($i \in [1, \ldots, d]$) is trained by using a dataset containing, for each key k in the seed map, an entry composed by the mapping of k in the feature space (obtained using k's FEATUREEXTRACTOR interface), and, as target class value, the i-th value associated with k in the seed map. As we are creating a decision-tree from scratch over a fully-known training set, we use, in this phase, VFDT as an offline-learner, by configuring it to use the C4.5 algorithm (Quinlan, 1993). This allows us to tightly control the cardinality of the ruleset it generates to achieve arbitrary accuracy in encoding the mapping, and, hence, fine tune the pruning of the ruleset to achieve the user specified parameter β. To this end, we need to keep into account that, when new data is added to the PAA (via the ADD method, to be discussed shortly), the decision tree is grown using the VFDT online algorithm, which can introduce an additional misclassification probability δ. Hence, during the pruning phase of the initial off-line trained decision tree, we aim at achieving a misclassification rate on the training set equal to $\beta' = 1 - \frac{1-\beta}{1-\delta}$.

The asymptotic complexity of the CREATE operation results as the sum of: i) creating an SBF — an operation that has constant cost, as it boils down to allocating a byte array whose size can be computed in O(1) based on α; ii) building the training set for d VFDTs via the FEATUREEXTRACTOR interface — an operation that has $O(S \cdot f \cdot d)$ cost, where S is the number of keys in the seed map, f is the number of features encoded in the keys, and d is the replication degree; and iii) training d VFDTs, which has complexity $O(d \cdot (S \cdot f \cdot log(S))$, which is the cost of building d decision trees over a training set of cardinality S with f attributes using C4.5 (Witten and Frank, 2005).

• GET: queries for a key k are performed by first querying the SBF. If the response is negative, $\perp$ is returned. Otherwise (and this may be a false positive with probability α), the

VFDT is queried by transforming k in to its representation in the feature space by means of the FeatureExtractor interface. If k had actually been inserted in the PAA, the query to the SBF is guaranteed to return a correct result. However, it may still be wrongly classified by the VFDT algorithm, which may return any of the target classes that it observed during the training phase. Despite such inaccuracies, operations will never be misrouted in AutoPlacer, since all nodes move data according to *LookupT*. At most, such inaccuracies may only lead to degraded performance, due to moving data that was not to be relocated, or due to moving data to unintended nodes.

The asymptotic complexity of the Get operation is equal to $O(h \cdot sl + dt - depth)$ where h and sl are, respectively, the number of hash functions and of internal Bloom filters used by the SBF, and $dt - depth$ is the depth of the decision tree induced by VFDT, which is typically assumed $O(log(S))$ when analyzing complexity of decision tree algorithms (Witten and Frank, 2005).

• Add: to implement this method, we leverage on the incremental features of the SBF and VFDT. To this end, we first insert each of the entries k passed as input into the SBF. This may lead to the generation of an additional, internal Bloom filter, to ensure that the bound on α. Next, we incrementally train the VFDT instances currently maintained in the PAA, by providing them, in a single batch, the entire set of key/value pairs that is being added to the PAA. As the VFDT algorithm introduces, at most, δ misclassification errors with respect to an off-line decision tree, and given that we prune the initial, off-line built decision tree to achieve accuracy β', we guarantee the tree built by VFDT upon the execution of the Add method still achieves the target bound on misclassification β.

The asymptotic complexity of the Add operation is equal to $O(Sh + dSf \ log(S))$, which corresponds to the cost of inserting S items in the SBF (by using h hash functions), plus the cost of build d trees using VFDT over a batch of S examples in the training set.

• getDelta: the output consists of the binary difference of the SBFs, plus the ruleset of the VFDT maintained by the PAA over which this method is invoked. To obtain the binary difference of the SBFs, we use a simple optimization that allows for avoiding the processing of all but the latest internal BF in common between the 2 PAAs being compared. As new elements are added always to the most recent internal BF, when this method is invoked over a PAA p, passing as input parameter a PAA p', it returns: i) any new BF included in p as a result of the insertion of additional elements in p', and ii) the most recent BF stored by p', which may have been in the meanwhile altered to store additional elements.

Excluding, for simplicity, this optimization from the analysis, the complexity of the GETDELTA operation can be estimated as proportional to the size (measured in bytes) of the PAA over which this operation is invoked.

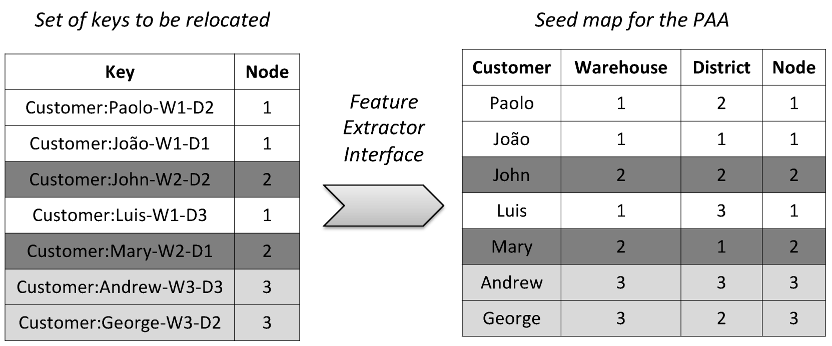
• APPLYDELTA: symmetrically to what is done in GETDELTA, this method generates a new PAA, whose SBF is obtained by applying the binary SBF difference contained in the input ΔPAA to the SBF of the PAA over which this method is invoked. The ruleset of the output PAA is set equal to the one contained in the input ΔPAA.

The complexity of this method, analogously to GETDELTA, is proportional to the size of the PAA passed as input.

4.4.4 Example use of PAAs in AutoPlacer

Figure 4.1 provides a concrete example of use of PAA in AUTOPLACER. Let us start by analyzing Figure 4.1(a). On the left side, we report an example set of keys whose placement in the system needs to be updated, together with their corresponding new placement. We recall that this information is obtained as output of the optimization phase of each AUTOPLACER's round. Before inserting these keys in the PAA, the FEATUREEXTRACTOR interface is first executed, yielding the result on the right. This phase allows extracting semantic information embedded in the key structure, and to map them into a feature space, which, in the considered example, consists of tuples of the form {CUSTOMER×WAREHOUSE×DISTRICT}, and having as target class the identifier of the node to which the key should be reassigned.

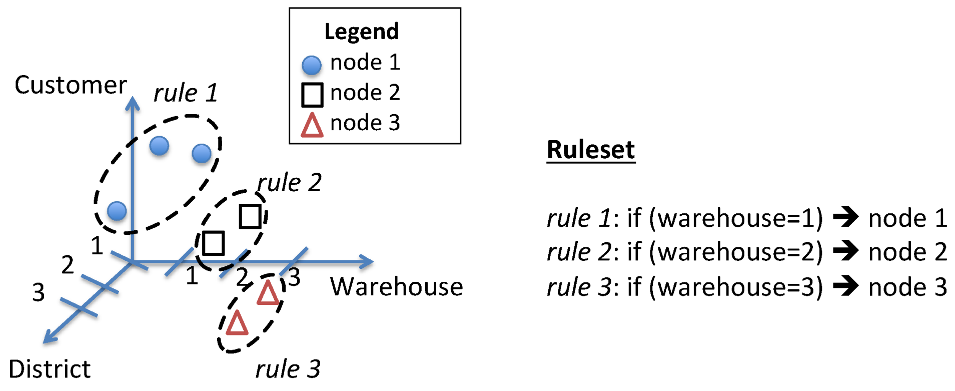
Figure 4.1(b) illustrates the mapping of the seed map in the feature space. Note that, in this diagram, we indicate with different point types (namely, a circle, a box, and a triangle) keys that are mapped to different nodes/values of the target class. Note also that we omit to label the CUSTOMER axis for the sake of readability. The figure also depicts a possible clustering induced by the ruleset determined by running the VDFT algorithm on the seed map. In the considered example, the decision tree is built using three rules that partition the feature space according to the value of the WAREHOUSE feature, and attribute a different target class value


Set of keys to be relocated
Feature Extractor Interface
Seed map for the PAA

Key	Node
Customer:Paolo-W1-D2	1
Customer:João-W1-D1	1
Customer:John-W2-D2	2
Customer:Luis-W1-D3	1
Customer:Mary-W2-D1	2
Customer:Andrew-W3-D3	3
Customer:George-W3-D2	3

Customer	Warehouse	District	Node
Paolo	1	2	1
João	1	1	1
John	2	2	2
Luis	1	3	1
Mary	2	1	2
Andrew	3	3	3
George	3	2	3

(a) Example use of the FEATUREEXTRACTOR Interface.



Ruleset

rule 1: if (warehouse=1) ➔ node 1
rule 2: if (warehouse=2) ➔ node 2
rule 3: if (warehouse=3) ➔ node 3

(b) Mapping of the seed map in the feature space and ruleset inference via VFDT.

Figure 4.1: Example use of PAAs in AUTOPLACER

(i.e., node) to each partition. In the considered example, which is clearly simplistic for the sake of clarity, the mapping of 7 keys to 3 different nodes (out of a possibly much larger set of total nodes) is encoded using only 3 rules, which can be encoded in a much more compact way than the relocation map. It should be noted that the considered ruleset may actually match a much larger number of keys than those included in the seed map. This would happen, for instance, each warehouse, in the considered example, was associated with a large population of customers. This is the reason why PAAs integrate a SBF, which they use to store the identifiers of the keys, whose mapping has been inserted in the PAA. Assume that a PAA, following its initialization with the considered set of keys, is queried about a key k not included in this set, where $k = Customer : Pedro - W1 - D2$. With high probability — more precisely, with the probability α specified upon the creation of the PAA — the key will not be present in the PAA's SBF, and the ruleset of the VFDT will not be queried. In this case, we recall that AUTOPLACER would consider the key not relocated, and locate it using the default consistent hash function.

On other hand, had the SBF suffered of a false positive, the PAA would have used the VFDT's ruleset to determine the value associated with k, and also suffer of a false positive.

An example of inaccurate mapping for a key stored in the PAA would have occurred in case the same ruleset had been produced with an input seed map containing also the key-value pair $< Customer : SomeName - W1 - D2, 2 >$. In this case, the ruleset induced by the VDFT algorithm misclassifies the key based on the fact that the value of its WAREHOUSE feature is 1, and erroneously associates it with the target value/node 1, instead of 2.

4.5 Optimizer Analysis

As already noted, the approximate nature of the information provided by top-k may affect the quality of the identified solution. An interesting question is therefore how degraded is the quality of the data placement solution when using top-k. Next, we provide an answer to this question by deriving an upper bound on the approximation ratio of the proposed algorithm in a single optimization round. Our proof shows that the approximation ratio is a function of the maximum approximation error provided by any *top-*$k_j(\mathcal{D})$, which we denote e^*, and of the average frequency of access to remote data items when using the optimal solution.

Theorem 1 The approximation ratio of the solution $\hat{X}$ found using the approximate frequencies $\hat{r}_{ik}$,$\hat{w}_{ik}$ is:

$$1 + \frac{d}{|\mathcal{N}| - d}\phi, \text{ with } \phi = \frac{e^*(cr^r + cr^w)}{cr^r rR + cr^w rW}$$

where e^* is the maximum overestimation error of top-k, and rR, resp. rW, is the average, across all nodes, of the number of read, resp. write, remote data items using the optimal data placement X^{Opt}.

Proof Let us now denote with $C(X, r_{ij}, w_{ij})$ the cost function used in Eq. 4.1 of the ILP formulation, restricted to the data items contained in *top-*$K(\mathcal{D})$:

$$\sum_{j \in \mathcal{N}} \sum_{i \in top\text{-}K(\mathcal{D})} \overline{X}_{ij}(cr^r r_{ij} + cr^w w_{ij}) + X_{ij}(cl^r r_{ij} + cl^w w_{ij})$$

and with $Opt = C(X^{opt}, r_{ij}, w_{ij})$ the value returned by the cost function using the binary matrix X^{opt} obtained solving the ILP problem with exact access statistics r_{ij}, w_{ij}.

Let lR, resp. rR, be the average, across all nodes, of the number of read accesses to local, resp. remote, data items using the optimal data placement X. lW and rW are analogously defined for write accesses. These can be directly computed, once known X^{Opt} and r_{ij},w_{ij} as:

$$
\begin{aligned}
rR &= \frac{\sum_{j\in\mathcal{N}}\sum_{i\in \mathit{top}\text{-}K(\mathcal{D})} \overline{X}_{ij}^{Opt} r_{ij}}{|\mathit{top}\text{-}K(\mathcal{D})|(|\mathcal{N}|-d)}\\
rW &= \frac{\sum_{j\in\mathcal{N}}\sum_{i\in \mathit{top}\text{-}K(\mathcal{D})} \overline{X}_{ij}^{Opt} w_{ij}}{|\mathit{top}\text{-}K(\mathcal{D})|(|\mathcal{N}|-d)}\\
lR &= \frac{\sum_{j\in\mathcal{N}}\sum_{i\in \mathit{top}\text{-}K(\mathcal{D})} X_{ij}^{Opt} r_{ij}}{|\mathit{top}\text{-}K(\mathcal{D})|d}\\
lW &= \frac{\sum_{j\in\mathcal{N}}\sum_{i\in \mathit{top}\text{-}K(\mathcal{D})} X_{ij}^{Opt} w_{ij}}{|\mathit{top}\text{-}K(\mathcal{D})|d}
\end{aligned}
$$

We can then rewrite Opt and derive its lower bound:

$$
\begin{aligned}
Opt =& |\mathit{top}\text{-}K(\mathcal{D})|((|\mathcal{N}|-d)(cr^r rR + cr^w rW)+\\
&+ d(cl^r lR + cr^w lW)) \geq\\
&\geq |\mathit{top}\text{-}K(\mathcal{D})|(|\mathcal{N}|-d)(cr^r rR + cr^w rW)
\end{aligned} \tag{4.2}
$$

Next, let us derive an upper bound on the "error" using the solution $\hat{X}$ obtained instantiating the ILP problem using the top-k-based frequencies $\hat{r}_{ij}, \hat{w}_{ij}$. The worst scenario is that an object $o \in \mathcal{D}$ is not assigned to the d nodes that access it most frequently because they do not include o in their top-k. In this case, we can estimate the maximum frequency with which o can have been accessed by any of these nodes as e^*. Hence, if we evaluate the cost function $C(\hat{X}, r_{ij}, w_{ij})$ using the exact data access frequencies, and the solution $\hat{X}$ of the ILP problem computed using approximate access frequencies, we can derive the following upper bound:

$$
C(\hat{X}, r_{ij}, w_{ij}) \leq Opt + |\mathit{top}\text{-}K(\mathcal{D})|de^*(cr^r + cr^w) \tag{4.3}
$$

The approximation ratio is therefore:

$$\frac{C(\hat{X}, r_{ij}, w_{ij})}{C(X^{Opt}, r_{ij}, w_{ij})} \leq 1 + \frac{d}{(|\mathcal{N}| - d)} \frac{e^*(cr^r + cr^w)}{cr^r rR + cr^w rW} \tag{4.4}$$

In the following corollary, we exploit the bounds on the space complexity of the Space-Saving Top-k algorithm (Metwally et al., 2005) to estimate the number of distinct counters to use, to achieve a target approximation factor $1 + \frac{d}{|\mathcal{N}|-d}\phi$.

Corollary 2 The number m of individual counters maintained by the Space-Saving Top-k algorithm to achieve an approximation factor equal to $1 + \frac{d}{|\mathcal{N}|-d}\phi$ is:

$$m = \frac{SL}{\phi} \frac{cr^r rR + cr^w rW}{cr^r + cr^w}$$

where SL is the total number of accesses in the stream.

Proof Derives from Theorem 6 of the work by Metwally et al., 2005, which introduced the Space-Saving Top-k algorithm. The Theorem states that to guarantee that the maximum over-estimation error $e^* \leq \epsilon F_k$, where F_k is the frequency of the k-th element in top-k, it is sufficient to use $m = \frac{SL}{\epsilon F_k}$ counters.

Finally, since in each round AutoPlacer optimizes the placement of a disjoint set of items, it follows that, if we assume stable data access distributions, the approximation ratio achieved by the optimization algorithm during round i will necessarily be lower (hence, better) than for round $i - 1$. In fact, at each round, the frequencies of the items tracked by the top-k will be lower than in the previous rounds, and, consequently, e^* will not increase over time.

4.6 Evaluation

This section evaluates the different aspects of the AutoPlacer. First, we describe our experimental setting. Next, we discuss the efficiency of the PAA data structure. We then present two case-studies of our system, one based on the well-known OLTP benchmark, TPC-C, and one based on a benchmark representative of geo-social applications, GeoGraph (Ziparo et al., 2013). In order to use both benchmarks in AutoPlacer, we modified the keys of objects in the benchmarks in order to implement the Feature Extractor Interface according to the static attributes of the objects they represent. For both benchmarks, we configured AutoPlacer to run new rounds every minute, and limited the $k = 1000$.

4.6.1 Experimental Settings

In order to evaluate experimentally AutoPlacer, we integrated it in the Infinispan key-value store. We have run experiments on two settings, one for each benchmark we make use of. For the TPC-C benchmark, we used a cluster of 40 virtual machines (deployed on 10 physical machines) running Xen, equipped with two 2.13 GHz Quad-Core Intel(R) Xeon(R) E5506 processors and 40 GB of RAM, running Ubuntu Linux 2.6.32-33-server and interconnected via a private Gigabit Ethernet. For the GeoGraph benchmark, we deployed our platform on of FutureGrid[1]. More specifically, we deployed the Cloud-TM Platform on top of a virtualized infrastructure with 25 VMs, which were configured with 16GB RAM, one 2.93GHz core Intel Xeon CPU X5570, running CentOS 5.5 x86 64 and interconnected by an InfiniBand network.

TPC-C Benchmark: Since Infinispan provides support for transactions, we developed for our experimental study a port of a well-known benchmark for transactional systems, namely the TPC-C benchmark (Leutenegger and Dias, 1993), which we adapted to execute on a key-value store[2]. TPC-C is a complex benchmark, which generates workloads representative of realistic OLTP environments, with complex and heterogeneous transactions having very skewed access patterns and high conflict probability. This is in contrast with common key-value store benchmarks (Cooper et al., 2010), which are typically composed of simple synthetic workloads.

Since our evaluation focuses on assessing the effectiveness of AutoPlacer in different scenarios of locality, we have modified the benchmark, so that we can induce controlled locality patterns in the data accesses of each node. This modification consists of configuring the benchmark so that the transactions originated on a given node access, with probability p data associated with a given warehouse, and, with probability $1-p$, data associated with a randomly

[1]http://portal.futuregrid.org

[2]The code of AutoPlacer and of the port of TPC-C employed in this evaluation study are freely available in the Cloud-TM project public repository: http://github.com/cloudtm

chosen warehouse. Thus, for example, by setting $p = 90\%$, nodes will have disjoint data access patterns (each accessing a different warehouse) for 90% of the transactions, while the remaining 10% access data uniformly. In the rest of the text, when we refer to "degree of locality" in the context of the TCP-C benchmark, we refer to the value of probability p as described above.

GeoGraph Benchmark: Geograph (Ziparo et al., 2013) is a benchmarking tool that allows injecting complex and rich workloads representative of the most popular geo-social applications. Geograph has been designed to be flexible both in the heterogeneity and in the dynamics of the workload. In particular, it provides 19 different geo-social services (i.e., actions) and 16 application specific user simulators. These simulators (named agents) can be combined in complex ways to generate dynamic workload profiles. In order to have a realistic geographical distribution of the agents, GeoGraph uses as baselines, for each simulation, real GPS traces from Open Street Maps[3].

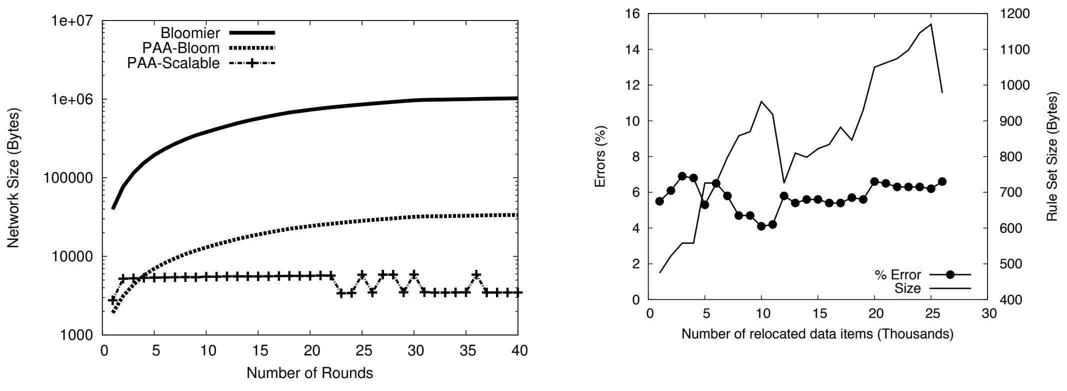
More specifically, the benchmark divides the spatial region where the agents move into $10km^2$ areas named Landmarks. It starts by inserting a set of posts (virtual messages) associated with each Landmark. Then, agents are spread throughout the map and move across it using realistic GPS traces, while issuing actions in its vicinities. The simplest action is READ-POST, which is used to query the landmark state for getting posted messages. The second action is UPDATE-POSITION which leads to a move operation of the agent between adjacent landmarks, and possibly to leave a new post.

In order to simulate a real-world deployment of a geo-social service, we have configured the benchmark to dispatch requests across the nodes in the system depending on their geographical origin. We then attributed geographical areas (sets of adjacent Landmarks) to servers, so that the requests of an agent are dispatched to the server responsible for the closest landmark, based on the agent's current geographic position.

4.6.2 Probabilistic Associative Array

In this section, we study tradeoffs in space efficiency and accuracy involved in the configuration and implementation of the PAA. For these results, we have used traces of the TPC-C

[3]http://www.openstreetmap.org/traces



(a) Traffic generated using different AAs (b) Error probability and ruleset size for VFDT

Figure 4.2: PAA Performance.

Table 4.3: Re-located objects and size of different PAA implementations.

PAA Implementation	**Re-located objects**	**Local space (KB)**
PAA-scalable	26600	150.8
PAA-Bloom	26600	31.84
Bloomier	26600	575.3

benchmark configured with 100% locality.

Bloom Filter Figure 4.2(a) presents the network bandwidth of different implementations of the PAA, compared with another form of probabilistic associative arrays, the Bloomier Filters (Bloomier) (Chazelle et al., 2004), as the rounds advance in the system. One implementation of the PAA uses regular Bloom filters (PAA-Bloom), while the other uses scalable Bloom filters (PAA-scalable). Both PAAs were configured with $\alpha = \beta = 0.01$, and the Bloomier filter's false positive probability was also set to 0.01. We note that the best solution is the one that allows to propagate in the network only differential updates with regard to previous state, which is the PAA-scalable. Concerning this latter solution, Figure 4.2(a) shows that in some rounds the message traffic generate by this solution oscillates. This can be explained considering that, as a result of the insertion of new data in the PAA, it may be possible to exhaust the capacity of the most recent internal BF of the scalable Bloom filter, and trigger the allocation of a new slice. In this case, the getDelta method of the PAA returns the last two (or potentially more, although this never occurred in this experiment) internal BFs.

Table 4.3 shows the local storage requirements at the end of the experiment. As it can be seen, PAA-SCALABLE has higher storage requirements than PAA-BLOOM. This is unsurprising, as scalable Bloom filters are known to achieve lower compression than traditional Bloom filters when fed with the same data set and configured to yield the same false positive rates (Almeida et al., 2007). However, the storage requirements of both solutions are still considerably smaller than those of Bloomier filters.

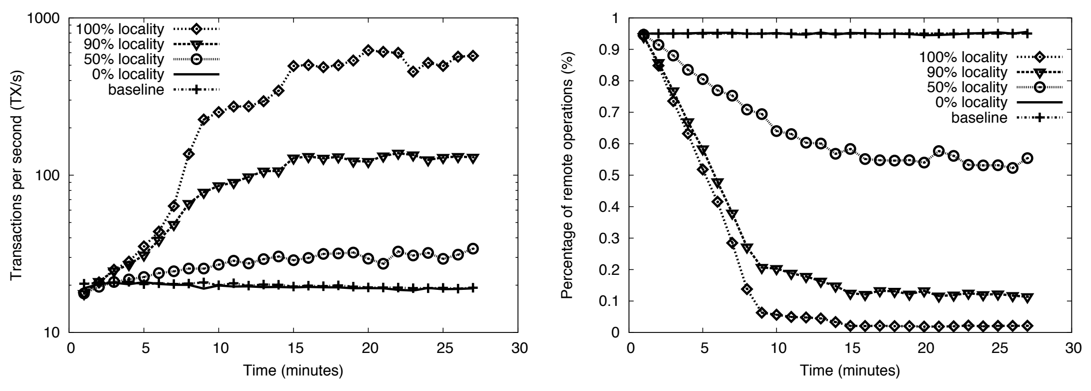
Machine Learner Figure 4.2(b) presents the error probability and space required by the DT to encode the objects moved in every round of the experiment. As more objects are moved in the system, the number of rules increases, leading to an increase in the size taken by this portion of the PAA. However, the machine learner can represent the mapping of 26600 keys in 1000 Bytes, corresponding to 213 rules. Furthermore, it can also be observed that while a significant number of keys is added to the machine learner (around 1000 per round), the error remains relatively stable.

4.6.3 Leveraging from Locality

This section shows how AUTOPLACER is able to leverage form locality patterns in the workload. The results were obtained with TPC-C, adapted as explained before and with a replication of degree $d = 2$.

Figure 4.3(a) shows the throughput of AUTOPLACER, compared with the non-optimized system using consistent hashing for different degrees of locality in the workload. In the baseline system, no matter how much locality exists in the workload, since consistent hashing is used to place the items, on average, the number of remote accesses does not change. Thus, for all workloads the baseline system exhibits a similar (sub-optimal) behaviour. In the system running AUTOPLACER, locality is leveraged by relocating data items. As times passes, and more rounds of optimization take place, system throughput increases up to a point where no further optimization is performed. It is interesting to note that, if there is no locality, throughput is not affected by the background optimization process. On the other hand, when locality exists, the throughput of the system optimized with AUTOPLACER is much higher than that of the baseline: it can be up to 6 times better for a workload with 90% locality, and up to 30 times better in the ideal case of 100% locality.

Figure 4.3(b) helps to understand the improvement in performance by looking at the number



(a) Throughput with varying degrees of locality. (b) % of remote operations.

Figure 4.3: AutoPlacer performance

of remote invocations that are performed in the system as time evolves. Since the initial setup relies on consistent hashing, both in the baseline and in the optimized system, the average probability of an operation being local is $\frac{1}{40} = 2.5\%$ for all workloads. Thus, when the system starts, most operations are remote. However, the plots clearly show that the number of remote operations decreases in time when using AutoPlacer. The plots also show another interesting aspect: although the number of remote operations decreases sharply after a few rounds of optimization, the overall throughput takes longer to improve. This is due to the fact that read transactions access a large number of objects, thus multiple rounds of optimization are required to alleviate the network, which is the bottleneck in these settings. This clearly highlights the relevance of the continuous optimization process implemented by AutoPlacer. At the end of the experiment, the percentage of operations performed locally is already close to the percentage of locality in the workload; this shows that when the system stabilizes, AutoPlacer was able to move practically all keys subject to locality.

Finally, Figure 4.4 compares the performance of AutoPlacer and that of a directory service-based system which may be used to store the mappings resulting from a global optimization in systems such as Ursa (You et al., 2013), or Schism (Curino et al., 2010) among other state-of-the-art solutions which lack an efficient way of broadcasting mappings. These results were obtained by storing the data relocation map obtained at the end of the entire optimization process into a dedicated directory service. In this case, whenever a node requests a data item that is not stored locally, it contacts the directory service to determine its location, instead of querying the local PAA. The results clearly show that the additional latency for contacting the

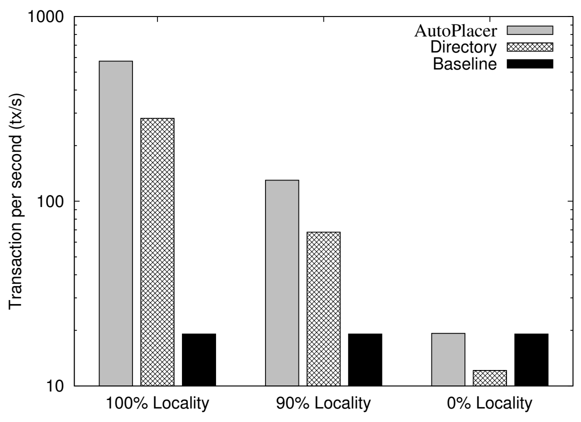


Figure 4.4: Throughput of AutoPlacer, a directory-based and a consistent hashing-based solution, after a complete optimization process.

directory service can hinder perform significantly, independently of the locality of the workload. The plots highlight that, unlike with AutoPlacer, the performance of directory-based systems can be worse than that achievable by using random placement. This is explainable considering that, with low locality, a large fraction of data accesses is remote, and that directory-based services impose a 2-hop latency, unlike consistent hashing and AutoPlacer. Note that, while the plot depicts throughput values, these numbers are highly correlated with the latency of individual operations; in fact, the decrease of throughput from AutoPlacer to that of a directory-based system is due to the latency of lookups and the decrease of throughput from the directory-based system to the baseline is due to the latency of (non-local) data accesses.

The speed-ups of AutoPlacer vs the directory-based solution are significant, i.e., around 2x, even for the high locality scenarios. In these scenarios, the reduction of the number of remote operations leads to less directory lookups. However, the cost of accessing a remote data item is, in our testbed, about 2 orders of magnitude larger than that of accessing a local item. As a consequence, also in these scenarios, the cost of remote data accesses dominates the execution time of the requests. Hence, such requests, which suffer from one additional communication hop latency in a directory-based solution, effectively limit the throughput of such a solution leading to considerably worse results than AutoPlacer.

4.6.4 Distributed Optimization

In this section, we explore how the design decisions behind AutoPlacer's distributed optimization algorithm affect the optimality of the final data placement. We have implemented a version of our optimizer which works in a centralized way using complete and precise traces of the system, obtained from all nodes during a long period. This is an approach similar to that of state-of-the-art solutions such as Ursa (You et al., 2013) and Schism (Curino et al., 2010). Finally, in order to improve the quality of the solution of the optimizer, so that it becomes an optimal adversary, we did not relax the ILP constraint in this version.

To achieve a fair comparison, we did three runs of TPC-C with the same settings as Section 4.6.3, and with two different configurations of Infinispan. Firstly, we did two long runs of an unmodified Infinispan, while collecting traces of data access. We used the traces from the first run to generate an optimization problem, which we fed to the centralized optimizer to derive an optimized placement. Next, we used the traces from the second run to generate another optimization problem, and store only its cost function. Finally, we ran AutoPlacer for the same period and collected the data placements resulting from each round of optimization. In this evaluation, we used the traces from the second run as a fair baseline for comparison between AutoPlacer and a centralized optimizer; We applied the data placement resulting from AutoPlacer and from the centralized optimizer to the cost function derived from the traces of the second run. This allowed us to compare the quality of the solution returned by each optimizer with respect to a run independent from that which the optimizers used as input.

Figure 4.5 shows how the quality of the solution generated by AutoPlacer evolves along rounds, as well as that of a centralized optimizer. This quality is a result of applying the cost function obtained from the second Infinispan run to the data placement solutions generated by each optimizer. The results are presented as a percentage of the cost of the (unoptimized) data placement generated by consistent hashing.

Similarly to the behaviour observed in Section 4.6.3, AutoPlacer converges progressively towards a steady-state, where the value for the objective function is not reduced any further. As expected, the more locality is encoded in the workload, the smaller is the value at which AutoPlacer converges, since more requests can be fulfilled by moving replicas to nodes requesting data items. In fact, the results shown in Figure 4.5 match those of Figure 4.3(b): since the remote operations are considerably more expensive than the local operations, the value of the

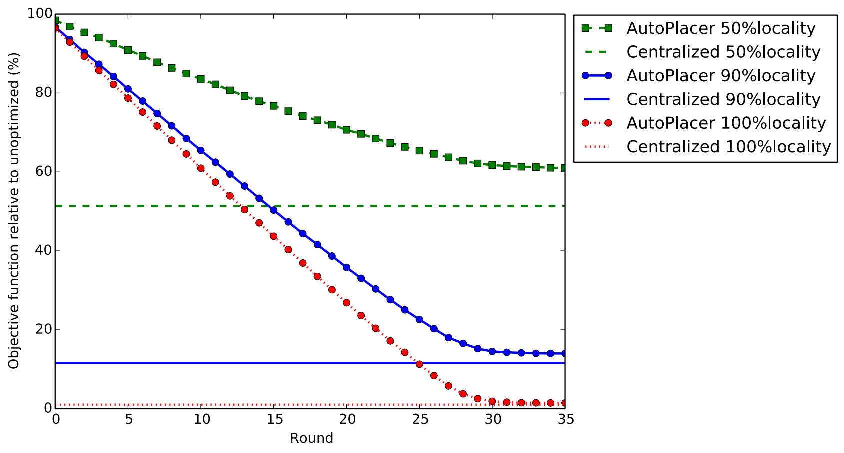


Figure 4.5: Progression of the quality of the data placement optimization solutions of AutoPlacer's distributed optimization versus that of a centralized optimizer, relative to unoptimized placement.

objective function at the end of the experiment relative to the unoptimized version corresponds roughly to the percentage of remote operations relative to the baseline in Figure 4.3(b).

Regarding the comparison with the centralized optimizer, it can be observed that AutoPlacer's optimizer's results are hindered by the fact that it is distributed and relies on uncertain information. However, it must be noticed that AutoPlacer optimizes the placement based on a much shorter window of time, which means that the statistics are more prone to be unreliable. This fact, combined with the fact that AutoPlacer avoids moving data more than once on each round, means that the system may make more wrong decisions. In absolute terms, this difference is more pronounced for lower locality scenarios, since AutoPlacer can more easily be deceived by the randomness in the workload than an algorithm which decides placement based on a longer period. In relative terms, the difference is more pronounced for higher locality scenarios, since any single deviation from the data placement devised by the centralized optimizer will have a strong impact on the objective function. Still, AutoPlacer was able to approach the results of the centralized optimizer to within 10% of the cost of the unoptimized placement, without requiring a complete trace of execution and while allowing the parallelization of the computation. Furthermore, AutoPlacer also provided considerable gains over the unoptimized placement, especially for the high-locality scenarios: the cost is reduced by up to 98.5%.

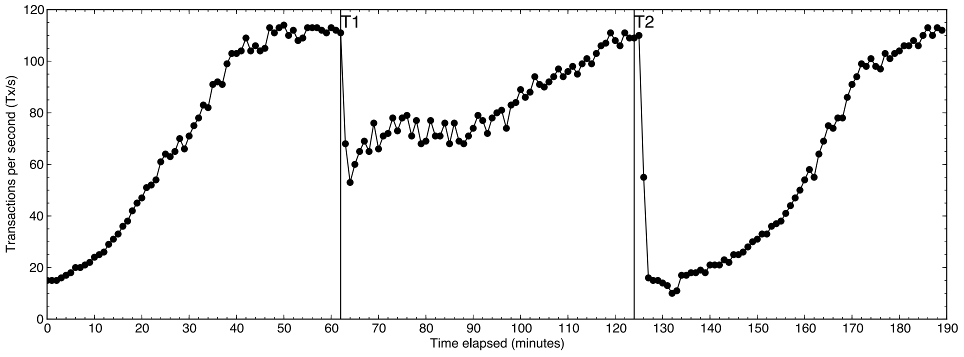


Figure 4.6: Total throughput of AutoPlacer over time for a dynamic workload with 90% of locality. T1 and T2 mark the instants when changes in workload occurred.

4.6.5 Dynamic workloads

In this section, we show how changes in the workload affect the behaviour of AutoPlacer. We deployed TPC-C using the same settings as in Section 4.6.3, with 90% access locality, but modified the benchmark to be able to introduce changes to the data access pattern at runtime. This allowed us to emulate the existence of a load balancer able to, at any time, decide to change the way in which requests are routed to each node (e.g., due to the change of popularity of some subset of data items).

Figure 4.6 presents the evolution of the throughput of the system when subject to two changes in workload, taking place at time T1 and T2. At T1, we cause a change in the data access pattern of 50% of the nodes. This change causes these nodes to access a different set of data, and to start accessing data which was previously being accessed by some other node in the set. At T2, we perform a similar change, to alter the data access pattern for all nodes in the system.

As Figure 4.6 shows, the throughput of the system drops considerably at both T1 and T2. This is caused by an increase in the remote read operations in the nodes affected by the change, which, not only cause their transactions to last longer, but also flood the network, affecting the transactions of the other nodes. It is noticeable, however, that the throughput drop at T1 is considerably smaller than at T2, since only 50% of the nodes are affected by the change. Furthermore, after T2, the throughput drops to roughly the same as at the beginning of the experiment, since after the change at T1, most of the read operations become remote as at the beginning of the experiment.

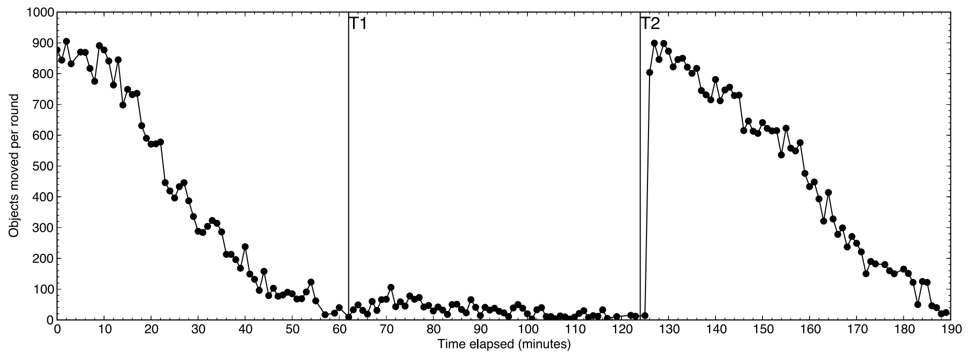


Figure 4.7: Number of objects moved to a specific node per round over time. The figure presents data for one of the nodes whose data is not affected by the first workload change (at T1).

In terms of convergence time, each section of the experiment runs for 64 minutes, but after roughly 40 minutes no significant increases in throughput are observed. It is important to notice that this fact also holds for the second section of the experiment (between T1 and T2). Even though the drop in performance after T1 is not as large as after T2, AutoPlacer still takes the same time to converge on the peak throughput as for the other sections. This happens because at each round, each node moves a limited number of objects, hence it will always require a similar number of rounds to optimize the same number of objects.

Figure 4.7 shows the number of objects moved to one of the nodes not affected by the change in workload triggered in T1. During the first rounds of optimization, the node receives most of its requested objects. As the system reaches a more stable throughput, the rate of objects moved per round is reduced. Notice also that, even though the node still receives around 80 objects per round after the throughput has stabilized, this object relocation leads to a marginal throughput gain. Between T1 and T2, this node's data remains roughly the same: since it is not affected by the workload change, it does not request any data from its neighbors. After T2, a new epoch is triggered, and since the node considered in Figure 4.7 is affected by the workload change, it starts receiving objects, until the system reaches a stable state towards the end of the experiment.

4.6.6 GeoGraph evaluation

This section studies the behaviour of AutoPlacer when using GeoGraph. Figure 4.8 presents the results for AutoPlacer running the GeoGraph benchmark. We started using the default, consistent-hashing data placement, with AutoPlacer disabled. After 10 minutes, we

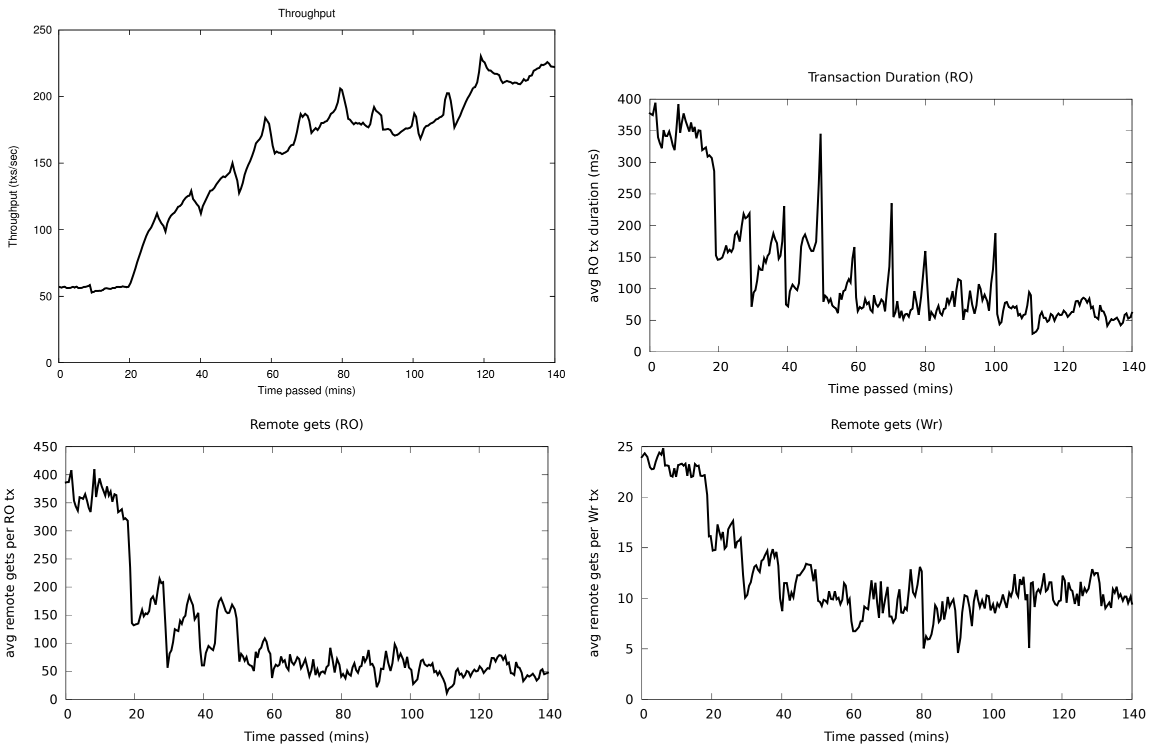


Figure 4.8: Evolution of throughput over time, duration of read-only transactions, and number of remote get operations per read-only and update transactions – GeoGraph benchmark.

activated AutoPlacer's statistics collection, and, after a period, to warm up the statistics, at minute 20, we triggered the first optimization epoch.

The plots in the top row of Figure 4.8 show how throughput and response time of read-only transactions vary over time during an experiment in which we inject load using 128 geographically dispersed agents, generating 90% read-only transactions (i.e., 90% READ-POST operations and 10% UPDATE-POSITION operations) with no think time. After approximately 10 minutes, when the performance of the system is of about 60 transactions per second, we trigger the gathering of statistics using the probabilistic top-k algorithm integrated in AutoPlacer. We notice that the throughput has a temporary drop of performance, with the throughput going down to 53 transactions over the following sampling interval, before achieving a stable throughput of about 57 transactions per second after about 5 minutes. The temporary drop in performance, immediately after the activation of the top-k tracing is most likely imputable to the initial overheads of the JVM in executing new code paths before the JIT targets them and optimizes them. Interestingly, however, the performance penalty introduced by top-k fades away almost completely over the following 5 minutes, where the system stabilizes its performance at about

5% lower than without top-k tracing.

By looking at the plots concerning the average number of remote get operations for both write and read-only transactions, in the bottom row of Figure 4.8, we can notice that transactions are generating a large amount of remote accesses, about 23 for write and about 360 for read-only transactions. This is a clear symptom of application's poor locality, which represents a typical use case that could potentially significantly benefit from AutoPlacer's locality-enhancing data-placement scheme.

We trigger AutoPlacer at around minute 20, and configure it to execute 10 rounds, collecting statistics for 1000 keys per node in each round. In order to assess the benefits on performance associated with the activation of each individual round, we set a conservative frequency of 1 activation every 10 minutes. This choice allows sufficient time to observe the dynamics of response of the system to the activation of the individual AutoPlacer's rounds. Analogously to the results obtained when using TPC-C, AutoPlacer provides significant benefits in terms of locality of the applications' data access patterns: over time, we see that the number of remote operations issued drops considerably, with the first rounds being the most effective. This phenomenon is explicable considering that, during the early optimization rounds, AutoPlacer relocates the hot-spot data items that are responsible for causing most of the remote accesses in each node, whereas, in the later rounds, the objects appearing in the nodes' top-k statistics are going to be accessed less and less frequently by the nodes. The last round ends at about 120 minutes from the beginning of the experiment.

At the end of the experiment, we concluded that the throughput fluctuates around 220 transactions per second. As expected, the throughput of AutoPlacer clearly increases over time, peaking at more than $4x$ of the initial, non-optimized throughput. This result is particularly good since AutoPlacer can infer the locality patterns of a realistic workload without requiring the programmer to know *a priori* how to partition the application's data and map it to the set of available nodes. The root causes of these gains are highlighted by the plots concerning the number of remote operations, as well as by the duration of read-only operations: both of them decrease significantly as subsequent rounds of AutoPlacer are executed, leading to a progressive efficiency growth of the system.

4.7 Summary

This chapter presented our solution for datacenter scale systems, which is based on moving selected data items to improve data locality. Despite supporting fine-grained placement of data items, AutoPlacer guarantees one hop routing latency through scalable mechanisms. The results showed that by taking advantage of high placement flexibility, AutoPlacer can achieve considerable throughput improvements over the state-of-the-art systems.

Notes

The results presented in this chapter were accomplished in cooperation with Pedro Ruivo and Paolo Romano. AutoPlacer was first proposed in the paper "AutoPlacer: scalable self-tuning data placement in distributed key-value stores", Proceedings of the 10th International USENIX Conference on Autonomic Computing, San Jose, CA, USA, June 2013. An extended version of the work was then published on the paper "AutoPlacer: scalable self-tuning data placement in distributed key-value stores", ACM Transactions on Autonomous and Adaptive Systems, Volume 9, Number 4, December 2014.

5 Final Remarks

Data placement is an important aspect of all data storage systems. It is equally relevant for in-memory storage or for systems that store data in persistent storage. Data placement affects several aspects of the system operation such as scalability, fault-tolerance, performance, among others. Several of these aspects have conflicting requirements, thus defining the optimal placement calls for a careful weight of these different aspects.

Unfortunately, even if an appropriate cost function can be defined for optimal placement, an arbitrary mapping between data items and nodes requires the maintenance of a directory service to locate the data replicas. Such services may be very expensive. Therefore, many practical systems have opted to use simple mapping functions, that may implement sub-optimal placements, but that can be very efficiently implemented.

In this thesis, we have presented two novel approaches to this problem. They explore a design space between simple mapping functions with small flexibility and expensive directories with full placement flexibility.

Our research advanced the state-of-the-art of flexibility for data placement in internet-scale systems, as well as scalability for data placement in datacenter-scale systems.

For internet-scale systems, moving nodes on the identifier space proved to be an effective way to increase placement flexibility. Rollerchain proposed a novel combination of gossip-based mechanisms and structured overlays to generate a DHT of virtual nodes, where each virtual node is materialized by a set of physical nodes. This design not only allows for a more flexible placement, but was also experimentally proved to be more robust than competing approaches. The increased flexibility provided by Rollerchain enabled us to design replication policies that can provide both low monitoring costs, low data transfer costs, and good load balancing. In particular, we proposed a novel policy, based on a principle that can be counter-intuitive at first: to put the most accessed data items on the less reliable nodes. We have shown that this policy, named "Hotter-On-Ephemeral", significantly outperforms previous work.

For the datacenter scale, individually moving select data items allowed AutoPlacer to optimize data placement in a more scalable way. AutoPlacer self-tunes data placement in a distributed key-value store. It operates in rounds, and, in each round, optimizes the placement of the top-k "hotspots", i.e., the objects generating most remote operations, for each node of the system. Despite supporting fine-grained placement of data items, AutoPlacer guarantees one-hop routing latency using a novel probabilistic data structure, the Probabilistic Associative Array, which minimizes the cost of maintaining and disseminating the data relocation map. AutoPlacer has been integrated in a popular open-source (transactional) key-value store, Infinispan, and experimentally evaluated using a porting of the TPC-C benchmark. Results show that AutoPlacer can achieve a throughput up to thirty times better than the original Infinispan implementation based on consistent hashing. Furthermore, its distributed optimization algorithm does not significantly hinder the optimization result.

Even though it might be desirable to find a unifying data placement strategy that would work well in both scenarios, our experience appears to indicate that such goal cannot be achieved. While Rollerchain provides more flexibility than previous approaches for internet-scale, it does not provide the granularity required by applications similar to AutoPlacer. On the other hand, while PAAs can provide significantly more flexibility than Rollerchain, they have to be broadcast to the whole network to update data placement, which makes them unsuitable for networks with high churn.

5.1 Research Collaborations

The insights provided by this thesis into how data placement strongly affects the performance of distributed storage systems led to additional research activities, which were pursued through collaborations. In the following, a brief summary of these results is presented:

Autonomic Configuration of HyperDex via Analytical Modelling

HyperDex (Escriva et al., 2012) is a recent multi-dimensional distributed key-value store that aims at supporting efficient queries using multiple objects' attributes. However, the advantage of supporting complex queries comes at the cost of a complex configuration. This work addresses the problem of automating the configuration of this innovative distributed indexing mechanism.

We derived a performance model that provides key insights on to how the placement of data affects the performance of the system. Based on this model, we derived a technique to automatically and dynamically select the best data placement by selecting one of several system configurations.

ShortMap

ShortMap is a system that relies on a combination of techniques aimed at efficiently supporting selective MapReduce jobs that are only concerned with a subset of the entire dataset. The system combines the use of an appropriate data layout with data indexing tools to improve the data access speed and significantly shorten the Map phase of the jobs. An extensive experimental evaluation of ShortMap shows that, by avoiding reading irrelevant blocks, it can provide speedups up to 80 times when compared to the basic Hadoop implementation. Further, the system also outperforms other MapReduce implementations that use variants of the techniques we have embedded in ShortMap.

5.2 Future Work

This research unveiled several new research avenues that could be pursued in the future:

- Despite the fact that AutoPlacer was proposed for improving data location, in fact, its distributed optimization algorithm and mappings broadcast through PAA might be applied to other goals. A prominent one would be to control the replication degree of items on a datacenter in a fine-grained but still scalable way. Since the current state-of-the-art systems require increasing the replication degree of several items at once (due to the usage of “buckets“ as described in Section 4.1), such an algorithm would drastically reduce the costs of replication.
- Despite this thesis proposing an efficient replication policy for internet-scale systems which takes advantage of Rollerchain's flexibility, it is clear that this avenue for research remains open. For instance, our work could be further improved by minding other system parameters such as communication locality, or arbitrary application-specific parameters.
- Since the PAA's design is generic, it can be decoupled from AutoPlacer and used as a stand-alone associative array. Thus, one possible research line is to use it to efficiently com-

municate other mappings in distributed systems. One example of such use is to broadcast load balancing directives (e.g. mappings of users to servers) between datacenter frontends of a storage layer.

Notes

The following lists the relevant publications that have resulted from the collaborative work discussed in this chapter.

“Auto-Configuração de Bases de dados NoSQL Multi-Dimensionais”, Nuno Diegues, Muhammet Orazov, João Paiva, Luís Rodrigues e Paolo Romano, Actas do Quinto Simpósio de Informática (Inforum), Évora, Portugal, September 2013

“Autonomic Configuration of HyperDex via Analytical Modelling”, Nuno Diegues, Muhammet Orazov, João Paiva, Luís Rodrigues and Paolo Romano, In Proceedings of the 29th Symposium On Applied Computing (ACM SAC’14), Gyeongju, South Korea, March 2014

“Optimizing Hyperspace Hashing via Analytical Modelling and Adaptation”,Nuno Diegues, Muhammet Orazov, João Paiva, Luís Rodrigues and Paolo Romano, In ACM SIGAPS Applied Computing Review (ACR),Volume 14, Number 2, June 2014

“Suporte eficiente para pesquisas seletivas em MapReduce”, Manuel Ferreira, João Paiva e Luís Rodrigues, Actas do Sexto Simpósio de Informática (Inforum), Porto, Portugal, September 2014

References

Abu-Libdeh, H., Geng, H., and van Renesse, R. (2011). Elastic replication for scalable consistent services (extended abstract). In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, SOSP'11, New York, NY, USA. ACM.

Agrawal, S., Narasayya, V., and Yang, B. (2004). Integrating vertical and horizontal partitioning into automated physical database design. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, SIGMOD'04, pages 359–370, New York, NY, USA. ACM.

Ahmad, M., Kemme, B., Brondino, I., Patiño-Martínez, M., and Jiménez-Peris, R. (2013). Transactional failure recovery for a distributed key-value store. In *Proceedings of the 14th IFIP/ACM International Conference on Distributed Systems Platforms*, Middleware'13, pages 267–286, Berlin Heidelberg. Springer-Verlag.

Almeida, P., Baquero, C., Preguiça, N., and Hutchison, D. (2007). Scalable bloom filters. *Information Processing Letters*, 101(6):255–261.

Alveirinho, J., Paiva, J., Leitão, J., and Rodrigues, L. (2010). Flexible and efficient resource location in large-scale systems. In *Proceedings of the 4th International Workshop on Large Scale Distributed Systems and Middleware*, LADIS'10, pages 55–60, New York, NY, USA. ACM.

Amza, C., Cox, A., and Zwaenepoel, W. (2003). Conflict-aware scheduling for dynamic content applications. In *Proceedings of the 4th Conference on USENIX Symposium on Internet Technologies and Systems*, USITS'03, Seattle, Washington, USA. USENIX Association.

Andrzejak, A. and Xu, Z. (2002). Scalable, efficient range queries for grid information services. In *Proceedings of the 2nd International Conference on Peer to Peer Computing*, P2P'02, pages 33–40, Washington, DC, USA. IEEE.

Ban, B. and Blagojevic, V. (2002). Reliable group communication with jgroups 3.x. Technical report, Red Hat, Inc.

Bauer, C. and King, G. (2006). *Java Persistence with Hibernate*. Manning Publications Co., Greenwich, USA.

Bishop, C. (2006). *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

Blake, C. and Rodrigues, R. (2003). High availability, scalable storage, dynamic peer networks: Pick two. In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems*, HotOS'03, pages 1–6, Lihue, Hawaii.

Blond, S., Fessant, F., and Merrer, E. (2009). Finding good partners in availability-aware P2P networks. In *Proceedings of the 11th International Symposium on Stabilization, Safety, and Security of Distributed Systems*, SSS '09, pages 472–484, Berlin, Heidelberg. Springer-Verlag.

Bloom, B. (1970). Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426.

Carter, J. B., Bennett, J. K., and Zwaenepoel, W. (1991). Implementation and performance of munin. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, SOSP'91, pages 152–164, New York, NY, USA. ACM.

Castro, M., Druschel, P., Kermarrec, A.-M., and Rowstron, A. (2006). Scribe: a large-scale and decentralized application-level multicast infrastructure. *Journal on Selected Areas in Communications*, 20(8):1489–1499.

Chandy, K. M. and Hewes, J. E. (1976). File allocation in distributed systems. In *Proceedings of the 1976 ACM SIGMETRICS Conference on Computer Performance Modeling Measurement and Evaluation*, SIGMETRICS '76, pages 10–13, New York, NY, USA. ACM.

Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E. (2008). Bigtable: a distributed storage system for structured data. *ACM Transactions on Compututer Systems*, 26(2):4:1–4:26.

Chazelle, B., Kilian, J., Rubinfeld, R., and Tal, A. (2004). The bloomier filter: an efficient data structure for static support lookup tables. In *Proceedings of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA'04, New Orleans, Louisiana, USA. Society for Industrial and Applied Mathematics.

Chen, H., Song, M., Song, J., Gavrilovska, A., and Schwan, K. (2011). Hears: A hierarchical energy-aware resource scheduler for virtualized data centers. In *Proceedings of the 2011 IEEE International Conference on Cluster Computing*, CLUSTER '11, pages 508–512, Washington, DC, USA. IEEE Computer Society.

Cook, N., Milojicic, D., and Talwar, V. (2012). Cloud management. *Journal of Internet Services and Applications*, 3(1):67–75.

Cooper, B., Silberstein, A., Tam, E., Ramakrishnan, R., and Sears, R. (2010). Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC'10, pages 143–154, New York, NY, USA. ACM.

Cooper, B. F., Ramakrishnan, R., Srivastava, U., Silberstein, A., Bohannon, P., Jacobsen, H.-A., Puz, N., Weaver, D., and Yerneni, R. (2008). Pnuts: Yahoo!'s hosted data serving platform. In *Proceedings of the International Conference on Very Large Databases*, VLDB'08, pages 1277–1288. VLDB Endowment.

Corbett, J. C., Dean, J., Epstein, M., Fikes, A., Frost, C., Furman, J. J., Ghemawat, S., Gubarev, A., Heiser, C., Hochschild, P., Hsieh, W., Kanthak, S., Kogan, E., Li, H., Lloyd, A., Melnik, S., Mwaura, D., Nagle, D., Quinlan, S., Rao, R., Rolig, L., Saito, Y., Szymaniak, M., Taylor, C., Wang, R., and Woodford, D. (2012). Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, OSDI'12, pages 251–264, Berkeley, CA, USA. USENIX Association.

Cruz, F., Maia, F., Matos, M., Oliveira, R., Paulo, J. a., Pereira, J., and Vilaça, R. (2013). Met: Workload aware elasticity for nosql. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 183–196, New York, NY, USA. ACM.

Curino, C., Jones, E., Zhang, Y., and Madden, S. (2010). Schism: a workload-driven approach to database replication and partitioning. In *Proceedings of the 36th International Conference on Very Large Databases*, VLDB'10, Singapore. VLDB Endowment.

Dabek, F., Kaashoek, M. F., Karger, D., Morris, R., and Stoica, I. (2001). Wide-area cooperative storage with cfs. In *Proceedings of the 8th ACM symposium on Operating systems principles*, SOSP'01, pages 202–215, New York, NY, USA. ACM.

DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. (2007). Dynamo: amazon's highly available key-value store. *SIGOPS Operating Systems Review*, 41(6):205–220.

Demmer, M. J. and Herlihy, M. (1998). The arrow distributed directory protocol. In *Proceedings of the 12th International Symposium on Distributed Computing*, DISC'98, pages 119–133, London, UK, UK. Springer-Verlag.

Didona, D., Romano, P., Peluso, S., and Quaglia, F. (2012). Transactional auto scaler: Elastic scaling of in-memory transactional data grids. In *Proceedings of the 9th ACM International Conference on Autonomic Computing*, ICAC'12, pages 125–134, New York, NY, USA. ACM.

Domingos, P. and Hulten, G. (2000). Mining high-speed data streams. In *Proceedings of the 6th International Conference on Knowledge Discovery and Data Mining*, SIGKDD'12, pages 71–80, New York, NY, USA. ACM.

Dowdy, L. and Foster, D. (1982). Comparative models of the file assignment problem. *ACM Computing Surveys*, 14(2):287–313.

Escriva, R., Wong, B., and Sirer, E. G. (2012). Hyperdex: A distributed, searchable key-value store. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM'12, pages 25–36, New York, NY, USA. ACM.

Fleisch, B. and Popek, G. (1989). Mirage: a coherent distributed shared memory design. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, SOSP'89, pages 211–223, New York, NY, USA. ACM.

Forell, T., Milojicic, D., and Talwar, V. (2011). Cloud management: Challenges and opportunities. In *IPDPS Workshops*, pages 881–889, New York. IEEE.

Garbatov, S. and Cachopo, J. (2011). Data access pattern analysis and prediction for object-oriented applications. *INFOCOMP Journal of Computer Science*, 10(4):1–14.

Ghodsi, A., Alima, L. O., and Haridi, S. (2007a). Symmetric replication for structured peer-to-peer systems. In *Proceedings of the 2005/2006 international conference on Databases, information systems, and peer-to-peer computing*, DBISP2P'05/06, pages 74–85, Berlin, Heidelberg. Springer-Verlag.

Ghodsi, A., Haridi, S., and Weatherspoon, H. (2007b). Exploiting the synergy between gossiping and structured overlays. *SIGOPS Operating Systems Review*, 41(5):61–66.

Glendenning, L., Beschastnikh, I., Krishnamurthy, A., and Anderson, T. E. (2011). Scalable consistency in scatter. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, SOSP'11, pages 15–28, New York, NY, USA. ACM.

Godfrey, B., Lakshminarayanan, K., Surana, S., Karp, R., and Stoica, I. (2004). Load balancing in dynamic structured p2p systems. In *Proceedings of the The 23rd Annual Joint Conference of the IEEE Computer and Communications Societies*, INFOCOM'04, pages 2253–2262, Washington, DC, USA. IEEE.

Gupta, I., Birman, K., Linga, P., Demers, A., and van Renesse, R. (2003). Kelips: Building an efficient and stable p2p DHT through increased memory and background overhead. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems*, IPTPS'03, Berkeley, CA, USA.

Harvey, N. J. A., Jones, M. B., Saroiu, S., Theimer, M., and Wolman, A. (2003). SkipNet: A Scalable Overlay Network with Practical Locality Properties. In *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems*, USITS'03, pages 9–9, Berkeley, CA, USA. USENIX.

Herlihy, M. and Sun, Y. (2005). Distributed transactional memory for metric-space networks. In *Proceedings of the 19th International Conference on Distributed Computing*, DISC'05, pages 324–338, Berlin, Heidelberg. Springer-Verlag.

Jelasity, M., Guerraoui, R., Kermarrec, A.-M., and van Steen, M. (2004). The peer sampling service: Experimental evaluation of unstructured gossip-based implementations. In *Proceedings of the 5th ACM/IFIP/USENIX International Conference on Middleware*, Middleware '04, pages 79–98, New York, NY, USA. Springer-Verlag New York, Inc.

Jia, Y., Brondino, I., Jiménez-Peris, R., Patiño Martínez, M., and Ma, D. (2013). A multi-resource load balancing algorithm for cloud cache systems. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, SAC'13, pages 463–470, New York, NY, USA. ACM.

Jiménez-Peris, R., Patiño Martínez, M., and Alonso, G. (2002). Non-intrusive, parallel recovery of replicated data. In *Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems*, SRDS'02, pages 150–159, Washington, DC, USA. IEEE.

Karger, D., Lehman, E., Leighton, T., Panigrahy, R., Levine, M., and Lewin, D. (1997). Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of 29th ACM Symposium on Theory of Computing*, STOC'97, pages 654–663, New York, NY, USA. ACM.

Kenthapadi, K. and Manku, G. S. (2005). Decentralized algorithms using both local and random probes for p2p load balancing. In *Proceedings of the 7th Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA'05, pages 135–144, New York, NY, USA. ACM.

Klemm, F., Girdzijauskas, S., Le Boudec, J.-Y., and Aberer, K. (2007). On routing in distributed hash tables. In *Proceedings of the 7th IEEE International Conference on Peer-to-Peer Computing*, P2P'07, Washington, DC, USA. IEEE.

Knežević, P., Wombacher, A., and Risse, T. (2009). Dht-based self-adapting replication protocol for achieving high data availability. In *Advanced Internet Based Systems and Applications*, pages 201–210, Berlin, Heidelberg. Springer-Verlag.

Knezevic, P., Wombacher, A., and Risse, T. (2005). Enabling high data availability in a DHT. In *The 16th Int'l Workshop on Database and Expert Systems Applications*, DEXA'05, Copenhagen, Denmark.

Krishnan, P., Raz, D., and Shavitt, Y. (2000). The cache location problem. *IEEE Transactions on Networking*, 8(7):568–582.

Lakshman, A. and Malik, P. (2010). Cassandra: a decentralized structured storage system. *SIGOPS Operating Systems Review*, 44(2):35–40.

Lamport, L. (1998). The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169.

Laoutaris, N., Telelis, O., Zissimopoulos, V., and Stavrakakis, I. (2006). Distributed selfish replication. *IEEE Transactions on Parallel and Distributed Systems*, 17(12):1401–1413.

Ledlie, J. and Seltzer, M. I. (2005). Distributed, secure load balancing with skew, heterogeneity and churn. In *Proceedings of the The 24rd Annual Joint Conference of the IEEE Computer and Communications Societies*, INFOCOM'05, pages 1419–1430, Washington, DC, USA. IEEE.

Leff, A., Wolf, J., and Yu, P. (1993). Replication algorithms in a remote caching architecture. *IEEE Transactions on Parallel and Distributed Systems*, 4(11):1185–1204.

Legtchenko, S., Monnet, S., Sens, P., and Muller, G. (2009). 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*, SSS'09, pages 485–499, Berlin, Heidelberg. Springer-Verlag.

Leitão, J. and Rodrigues, L. (2014). Overnesia: a resilient overlay network for virtual superpeers. In *Proceedings of the 33rd International Symposium on Reliable Distributed Systems*, SRDS'14, Washington, DC, USA. IEEE.

Leutenegger, S. and Dias, D. (1993). A modeling study of the tpc-c benchmark. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, SIGMOD'93, pages 22–31, New York, NY, USA. ACM.

Li, K. and Hudak, P. (1989). Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359.

Li, S., Abdelzaher, T., and Yuan, M. (2011). Tapa: Temperature aware power allocation in data center with map-reduce. In *Proceedings of the 2011 International Green Computing Conference and Workshops*, IGCC'11, pages 1–8, Washington, DC, USA. IEEE Computer Society.

Li, S., Wang, S., Yang, F., Hu, S., Saremi, F., and Abdelzaher, T. (2013). Proteus: Power proportional memory cache cluster in data centers. In *Proceedings of the 33nd International Conference on Distributed Computing Systems*, ICDCS'13, pages 73–82, Washington, DC, USA. IEEE Computer Society.

Lim, B.-H., Chang, C.-C., Czajkowski, G., and von Eicken, T. (1997). Performance implications of communication mechanisms in all-software global address space systems. In *Proceedings of the Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP'97, pages 230–239, New York, NY, USA. ACM.

Liu, G. and Maguire, G. (1994). *A Survey of Caching and Prefetching Techniques in Distributed Systems*. Trita-IT. Department of Teleinformatics.

Liu, H. and Motoda, H. (1998). *Feature Selection for Knowledge Discovery and Data Mining*. Kluwer Academic Publishers, Norwell, MA, USA.

Lynch, N. A., Malkhi, D., and Ratajczak, D. (2002). Atomic data access in distributed hash tables. In *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, IPTPS '01, pages 295–305, London, UK, UK. Springer-Verlag.

Malkhi, D., Naor, M., and Ratajczak, D. (2002). Viceroy: a scalable and dynamic emulation of the butterfly. In *Proceedings of the 21st Annual Symposium on Principles of Distributed Computing*, PODC'02, pages 183–192, New York, NY, USA. ACM.

Maniymaran, B., Bertier, M., and Kermarrec, A.-M. (2007). Build one, get one free: Leveraging the coexistence of multiple p2p overlay networks. In *Proceedings of the 27th International Conference on Distributed Computing Systems*, ICDCS'07, page 33, Washington, DC, USA. IEEE.

Marchioni, F. and Surtani, M. (2012). *Infinispan Data Grid Platform.* PACKT Publishing.

Maymounkov, P. and Mazières, D. (2002). Kademlia: A peer-to-peer information system based on the xor metric. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems*, IPTPS'02, Cambridge, MA, USA.

Metwally, A., Agrawal, D., and El Abbadi, A. (2005). Efficient computation of frequent and top-k elements in data streams. In *Proceedings of the 10th International Conference on Database Theory*, ICDT'05, pages 398–412, Edinburgh,Scotland. Springer-Verlag.

Mickens, J. and Noble, B. (2006). Exploiting availability prediction in distributed systems. In *Proceedings of the 3rd Conference on Networked Systems Design & Implementation*, NSDI'06, Berkeley, CA, USA. USENIX Association.

Mitchell, T. (1997). *Machine Learning.* McGraw-Hill, New York, NY, USA.

Montresor, A. and Jelasity, M. (2009). PeerSim: A scalable P2P simulator. In *Proceedings of the 9th International Conference on Peer-to-Peer*, P2P'09, pages 99–100, Washington, DC, USA. IEEE.

Moon, B., Jagadish, H. v., Faloutsos, C., and Saltz, J. H. (2001). Analysis of the clustering properties of the hilbert space-filling curve. *IEEE Transactions on Knowledge and Data Engineering*, 13(1):124–141.

Pace, A., Quéma, V., and Schiavoni, V. (2011). Exploiting node connection regularity for DHT replication. In *Proceedings of the 30th International Symposium on Reliable Distributed Systems*, SRDS'11, pages 111–120, Washington, DC, USA. IEEE.

Paiva, J., Leitao, J., and Rodrigues, L. (2013). Rollerchain: A DHT for efficient replication. In *Proceedings of the 12th IEEE International Symposium on Network Computing and*

Applications, NCA'13, pages 17–24, Washington, DC, USA. IEEE Computer Society.

Patterson, D. A., Gibson, G., and Katz, R. H. (1988). A case for redundant arrays of inexpensive disks (raid). In *Proceedings of the 1988 ACM SIGMOD international conference on Management of data*, SIGMOD'88, pages 109–116, New York, NY, USA. ACM.

Pavlo, A., Curino, C., and Zdonik, S. (2012). Skew-aware automatic database partitioning in shared-nothing, parallel oltp systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, SIGMOD'12, pages 61–72, New York, NY, USA. ACM.

Peluso, S., Romano, P., and Quaglia, F. (2012a). Score: A scalable one-copy serializable partial replication protocol. In *Proceedings of the 13th IFIP/ACM International Conference on Distributed Systems Platforms*, Middleware'12, pages 456–475, New York, NY, USA. Springer-Verlag New York, Inc.

Peluso, S., Ruivo, P., Romano, P., Quaglia, F., and Rodrigues, L. (2012b). When scalability meets consistency: Genuine multiversion update-serializable partial data replication. In *Proceedings of the 32nd International Conference on Distributed Computing Systems*, ICDCS'12, pages 455–465, Piscataway, NJ, USA. IEEE.

Quinlan, J. R. (1993). *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Ratnasamy, S., Francis, P., Handley, M., Karp, R., and Shenker, S. (2001). A scalable content-addressable network. In *Proceedings of the annual conference of the Special Interest Group on Data Communication*, SIGCOMM'01, pages 161–172, New York, NY, USA. ACM.

RedHat/JBoss (2013). Non blocking state transfer v2. https://github.com/infinispan/infinispan/wiki/Non-Blocking-State-Transfer-V2.

Romano, P., Little, M., Quaglia, F., Rodrigues, L., and Ziparo, V. (2014). Cloud-tm: Transactional, object-oriented, self-tuning cloud data store. Technical Report 7, INESC-ID.

Rowstron, A. and Druschel, P. (2001a). Storage management and caching in past, a large-scale, persistent peer-to-peer storage utility. In *Proceedings of the 8th ACM symposium on Operating systems principles*, SOSP'01, pages 188–201, New York, NY, USA. ACM.

Rowstron, A. I. T. and Druschel, P. (2001b). Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proceedings of the IFIP/ACM Inter-*

national Conference on Distributed Systems Platforms, Middleware '01, pages 329–350, Berlin, Heidelberg. Springer-Verlag.

Ruivo, P., Couceiro, M., Romano, P., and Rodrigues, L. (2011). Exploiting total order multicast in weakly consistent transactional caches. In *Proceedings of the the 17th Pacific Rim International Symposium on Dependable Computing*, PRDC'11, Piscataway, NJ, USA. IEEE.

Sangyeol, L. and Taewook, L. (2004). Cusum test for parameter change based on the maximum likelihood estimator. *Sequential Analysis: Design Methods and Applications*, 23(2):239–256.

Schmidt, C. and Parashar, M. (2004). Enabling flexible queries with guarantees in p2p systems. *IEEE Internet Computing*, 8(3):19–26.

Schutt, T., Schintke, F., and Reinefeld, A. (2006). Structured overlay without consistent hashing: Empirical results. In *Proceedings of the 6th Workshop on Global and Peer-to-Peer Computing*, GP2PC'06, pages 8–8, Washington, DC, USA. IEEE.

Schütt, T., Schintke, F., and Reinefeld, A. (2008). Scalaris: reliable transactional p2p key/-value store. In *Proceedings of the 7th ACM SIGPLAN workshop on ERLANG*, pages 41–48, New York, NY, USA. ACM.

Shafaat, T. M., Ahmad, B., and Haridi, S. (2012). Id-replication for structured peer-to-peer systems. In *Proceedings of the 18th International Conference on Parallel Processing*, Euro-Par'12, pages 364–376, Berlin, Heidelberg. Springer-Verlag.

Stoica, I., Morris, R., Karger, D., Kaashoek, M. F., and Balakrishnan, H. (2001). Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the annual conference of the Special Interest Group on Data Communication*, SIGCOMM'01, pages 149–160, New York, NY, USA. ACM.

Tatarowicz, A. L., Curino, C., Jones, E., and Madden, S. (2012). Lookup tables: Fine-grained partitioning for distributed databases. In *Proceedings of the 28th International Conference on Data Engineering*, ICDE'12, pages 102–113, Washington, DC, USA. IEEE Computer Society.

van Renesse, R. and Schneider, F. B. (2004). Chain replication for supporting high throughput and availability. In *Proceedings of the 6th conference on Symposium on Opearting Systems*

Design & Implementation, OSDI'04, pages 7–7, Berkeley, CA, USA. USENIX Association.

Vilaça, R., Oliveira, R., and Pereira, J. (2011). A correlation-aware data placement strategy for key-value stores. In *Proceedings of the 11th IFIP international conference on Distributed applications and interoperable systems*, DAIS'11, pages 214–227, Berlin, Heidelberg. Springer-Verlag.

Waldvogel, M., Hurley, P., and Bauer, D. (2003). Dynamic replica management in distributed hash tables. Technical Report RZ–3502, IBM.

Wang, L., Xu, J., Zhao, M., and Fortes, J. (2011). Adaptive virtual resource management with fuzzy model predictive control. In *Proceedings of the 8th ACM International Conference on Autonomic Computing*, ICAC'11, pages 191–192, New York, NY, USA. ACM.

Witten, I. and Frank, E. (2005). *Data Mining: Practical Machine Learning Tools and Techniques, Second Edition (Morgan Kaufmann Series in Data Management Systems).* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Wu, D., Tian, Y., and Ng, K.-W. (2006). Analytical study on improving DHT lookup performance under churn. In *Proceedings of the 6th IEEE International Conference on Peer-to-Peer Computing*, P2P'06, Washington, DC, USA. IEEE.

You, G.-W., Hwang, S.-W., and Jain, N. (2013). Ursa: Scalable load and power management in cloud storage systems. *ACM Transactions on Storage*, 9(1):1:1–1:29.

Zaman, S. and Grosu, D. (2011). A distributed algorithm for the replica placement problem. *IEEE Transactions on Parallel and Distributed Systems*, 22(9):1455–1468.

Zhuang, S. Q., Zhao, B. Y., Joseph, A. D., Katz, R. H., and Kubiatowicz, J. D. (2001). Bayeux: an architecture for scalable and fault-tolerant wide-area data dissemination. In *Proceedings of the 11th NOSSDAV*, pages 11–20, New York, NY, USA. ACM.

Ziparo, V., Cottefoglie, F., Calisi, D., Zaratti, M., Giannone, F., and Romano, P. (2013). D4.3 - prototype of pilot application i. In *Cloud-TM project.*