# Decrepol: A Decentralized Replication Policy for Web Documents

Paul Zijlmans
pzijlma@cs.vu.nl
Student number: 1143336

Computer Science
Master Thesis

Vrije Universiteit Amsterdam
Department of Computer Science

Supervisor:
Guillaume Pierre

August 24, 2006

**Abstract**

The current size of the Internet makes it impossible to host major Web sites on a single server. Replication can improve both the availability and the performance of a Web hosting service. Current replication policies heavily rely on the origin server. Therefore, this thesis presents Decrepol, a decentralized replication policy for Web documents that assumes the origin server to be unavailable most of the time. First, we present a replication policy that achieves full replication and is structured following epidemic protocols. With this policy we are able to spread notifications across 128 replica servers in about 5 gossip rounds. Next, we present a decentralized replication policy that allows for controlled partial replication. We create two versions of the partial replication policy. The first one is based on epidemic protocols and is an extented version of the full replication policy. However, in this policy locating replicas becomes a problem. Therefore, we also present an improved unstructured version that facilitates locating content. With this version, inserting and retrieving documents requires traversing about three hops. The second version is based on structured peer-to-peer systems. Inserts take about four hops, retrieves only two hops if we cache document searches. We show that it is possible to create an unstructured partial replication policy that performs almost identically to a structured one and thus can be an interesting alternative.

**Keywords:**

Replication policy, Web hosting, Peer-to-peer, Decentralization.

# Contents


**1 Introduction** **3**

**2 Related Work: Replication Policies for Web Documents** **5**
- 2.1 Number of replicas . . . 5
- 2.2 Replica placement . . . 6
- 2.3 Consistency enforcement . . . 8
  - 2.3.1 Consistency models . . . 8
  - 2.3.2 Content distribution mechanisms . . . 9
- 2.4 Request routing . . . 11
- 2.5 Availability . . . 11
- 2.6 Replication policies in Globule . . . 12

**3 Related Work: Peer-to-Peer** **14**
- 3.1 Overview . . . 14
- 3.2 Unstructured peer-to-peer systems based on epidemic protocols . . . 15
- 3.3 Structured peer-to-peer systems . . . 19

**4 A Full Replication Policy** **23**
- 4.1 The problem . . . 23
  - 4.1.1 Server selection . . . 24
  - 4.1.2 Select items to send . . . 24
  - 4.1.3 Select items to keep . . . 25
  - 4.1.4 Probability functions . . . 25
- 4.2 Performance evaluation . . . 26
  - 4.2.1 Select items to send (infinite cache size) . . . 27
  - 4.2.2 Select items to send (finite cache size) . . . 27
  - 4.2.3 Select items to keep . . . 28
  - 4.2.4 Threshold . . . 30
  - 4.2.5 Optimal cache size and gossip length . . . 32
- 4.3 Conclusion . . . 34

**5 A Partial Replication Policy** **36**
- 5.1 Unstructured version . . . 37
  - 5.1.1 Replica placement . . . 37
  - 5.1.2 Consistency enforcement . . . 37
  - 5.1.3 Replica location . . . 37
  - 5.1.4 Availability . . . 38
  - 5.1.5 Maintenance . . . 38
- 5.2 Performance evaluation unstructured version . . . 38
  - 5.2.1 Requests . . . 39
- 5.3 Improved unstructured version . . . 39

5.4 Performance evaluation improved unstructured version . . . 40
5.4.1 Inserts . . . 40
5.4.2 Requests . . . 41
5.4.3 Costs . . . 42
5.5 Structured version . . . 42
5.5.1 Replica placement . . . 42
5.5.2 Consistency enforcement . . . 43
5.5.3 Replica location . . . 44
5.5.4 Availability . . . 44
5.5.5 Maintenance . . . 44
5.6 Performance evaluation structured version . . . 45
5.6.1 Requests . . . 45
5.6.2 References . . . 47
5.6.3 Inserts . . . 49
5.6.4 Costs . . . 49
5.7 Comparison . . . 49

**6 Conclusion** **51**

**Bibliography** **53**

# Chapter 1

# Introduction

The current size of the Internet makes it impossible to host major Web sites on a single server. As Web servers may fail and parts of the network can be overloaded, a Web server can easily become unavailable. To prevent a Web site of becoming unreachable, one can use *replication*. By distributing multiple copies of documents at well-chosen locations, *replica servers*, one can improve both the availability and the performance of the Web hosting service. Availability increases as clients can switch to another replica when a server fails. Placing a replica of a data item hosted by a heavily loaded server on a server with a lower workload and subsequently dividing the workload between these two servers can also improve performance. Furthermore, placing replicas in the proximity of clients can reduce client-perceived latencies.

However, having multiple copies of a document introduces a consistency problem. When one updates a copy of a data item, the other copies have to be destroyed or brought up-to-date as well to prevent clients from retrieving stale documents. Other issues that we have to take into account when we replicate Web documents are the number of replicas of a document, where we place them, how we route a request for a document to a server hosting a replica, and how we ensure availability in the presence of failures. We define a *replication policy* as a set of algorithms that takes care of these issues.

Current replication policies heavily rely on the *origin server*, which we define as the server that hosts the original versions of the Web documents in the system. The origin is the only server that can update these documents. It is also responsible for placing the right number of replicas at suitable replica servers and keeping them consistent. A problem of this architecture is that the origin server becomes a single point of failure. A basic solution for this problem is to use one or more backup servers that all contain a full copy of the documents of the origin server and which one can contact in case the origin server is unavailable. However, one needs to keep these backup servers consistent.

Another issue of current replication policies is that the origin server has to be online all the time. When an origin server is temporarily unavailable, backup servers can cover that, but these backup servers are in fact full copies of the origin server.

Finally, when a replica server becomes unavailable, the origin server cannot inform this replica server about a document update. However, when the replica server recovers it needs to receive the update messages it missed during the failure in order to update its documents. As the origin is responsible for keeping the replicas consistent, it has to buffer the update messages until the replica server becomes available again.

The goal of this thesis is to design a decentralized replication policy for Web documents. The origin server does not have to be online all the time and can for example be a laptop or a PDA. We need to organize the replica servers in a decentralized fashion such that at least one of them has a copy of each document. If a client requests a replica server for a document the server does not possess locally, this server should know where to fetch it from. Replica servers can also fail. The replication policy should therefore be tunable such that copies are available

at $n$ different replica servers. This way the replication policy supports up to $n-1$ unavailable replica servers. Finally, the replication policy must implement best-effort weak consistency by ensuring that updates are spread in a reasonable limited time interval after an update takes place at the original document.

An important feature of this replication policy is that it can assume the origin server to be unavailable most of the time. It only needs the presence of the origin server to inform the replica servers about a new or updated document. The origin server can achieve this by just sending a notification to one of the replica servers and subsequently waiting until at least one replica server fetches the document. The replica servers are responsible of further spreading the notification among themselves.

The properties of this decentralized replication policy come close to those offered by peer-to-peer overlays: sharing computer resources, decentralization, self-organization, resilience to network and server failures. Therefore, we decided to structure the replication policy along a peer-to-peer architecture. We first introduce a replication policy that achieves *full replication*, which means that every replica server hosts a copy of all documents of the origin server. This is a simple version of a replication policy that we use as the base of a replication policy that achieves a more general form of replication, namely *partial replication*. The main challenge for the full replication policy is to spread the notifications across all replica servers in an efficient and decentralized way. These requirements are very similar to the properties of epidemic protocols. Therefore, we structure the full replication policy following epidemic protocols.

Next, we present a decentralized replication policy that allows for controlled partial replication. With controlled partial replication documents are replicated to exactly $k$ replica servers in a network with $N$ nodes and $0 < k \leq N$. We can distinguish a number of issues of the partial replication policy. First, the policy has to make sure that it places replicas at exactly $k$ replica servers. Second, it needs to keep the replicas of a document consistent in the presence of updates, as we may place replicas of an updated document at other replica servers than the replicas of the previous version of the document. Third, the policy must be able to locate replicas, as clients may request documents from replica servers that do not have a local copy of the requested document. Finally, the policy has to deal with node and network failures.

We create two versions of the partial replication policy. The first one is based on epidemic protocols and is an extended version of the full replication policy. The advantage of epidemic protocols is that they are easily maintained. However, as there is no correlation between nodes and the content, locating content may become difficult. In contrast, the second version is based on structured peer-to-peer systems. An advantage of these structured systems is that they provide a scalable solution for efficiently routing queries to the node with the desired content. However, one needs to maintain the structure (required for efficiently routing messages) in the presence of nodes joining and leaving the system. We compare the two versions and identify their strong and weak properties.

This thesis is structured as follows. In Chapter 2 we discuss the issues of replication policies for Web documents. Chapter 3 gives an overview of peer-to-peer systems and describes how we can use peer-to-peer technology for our decentralized replication policy. In Chapter 4 we present and evaluate the full replication policy; Chapter 5 presents and evaluates the unstructured and structured versions of the partial replication policy. Finally, Chapter 6 concludes.

# Chapter 2

# Related Work: Replication Policies for Web Documents

Achieving good performance with distributed systems in large computer networks such as the Internet can be hard. Performance bottlenecks are usually due to failing or poorly performing servers and overloaded parts of the network.

*Replication* is a technique that can be used to improve the quality of distributed services. By distributing multiple copies of data, *replicas*, at well-chosen locations, *replica servers*, one can improve both the availability and the performance of the service [24, 15]. Availability increases as clients can switch to another replica when a server fails. Creating a copy of a data item hosted by a heavily loaded server on a server with a lower load and subsequently dividing the workload between the two servers can also improve performance. Furthermore, placing replicas in the proximity of the clients can reduce client-perceived latencies.

Although replication can solve scalability problems, having multiple data copies introduces a consistency problem. When one updates a copy of a data item, the other copies have to be destroyed or brought up-to-date as well to prevent clients from retrieving stale data. Unfortunately, maintaining this consistency is often costly.

Web site hosting is a distributed service where replication has been increasingly applied in the last few years [22]. Placing replicas of a site's Web documents at replica servers can improve the accessibility of the Web site. There are many ways to replicate a Web document across multiple servers. One must decide how many copies of the document are needed, where to create them and how to keep them consistent. Furthermore, a decision is needed on how to route a request for a document to a server hosting a replica. Another issue that one has to considere is how to ensure availability when failures occur. In this thesis we define a *replication policy* as a set of algorithms that makes these decisions.

Next, we give an overview of the research that has been done on these replication policy issues. Section 2.1 describes which issues determine the number of necessary replicas of a document. Section 2.2 discusses the problem of finding good locations to host replicas of a document. Section 2.3 describes how one can keep multiple copies of a document consistent. We discuss request routing in Section 2.4. Section 2.5 discusses availability. Finally, Section 2.6 gives an overview of the current replication policies in Globule and states what the proposed replication policy adds to them.

## 2.1 Number of replicas

The number of necessary replicas of a certain Web document depends on two issues: the requested degree of fault tolerance and the requested client-perceived performance. Furthermore, this number is often bounded by storage and administrative constraints.

Fault tolerance affects the decision of how many replicas are needed, as at least $n+1$ replicas are needed to support $n$ failing replicas. This way, a client can switch to at least one correctly working replica when up to $n$ replicas are unavailable. Consequently, an increase in the number of replicas increases the reliability of the document.

Another issue is the client-perceived performance, i.e. the communication delay experienced by the clients. One needs to consider two things when determining the number of replicas to achieve a given performance: the popularity of the document and the document's client locations. Popular documents are requested more often and generate more workload on the hosting server. By creating several replicas at different servers this load can be spread across these servers. Besides the request rate of a document, the locations of the requesting clients can also influence the required number of replicas. When clients are located in different parts of the Internet, deploying several replicas in these areas and redirecting each client to its proximal replica can improve the communication delay between the client and the replica. Ideally, one should place replicas in those areas where the document is most popular.

Unfortunately, deploying replicas does not come for free. One has to create all replicas at a replica server. Besides the storage costs, there are also administrative costs, as one has to keep the replicas consistent. This introduces global communication. We discuss consistency enforcement in more detail in Section 2.3. The decision of how many replicas one needs for a certain document is a tradeoff between reliability and performance on the one hand, and the storage and administrative costs on the other hand. This decision is up to the owner of the document and depends on his requirements. The goal of a good replication policy is, when one has determined the number of replicas, to take care of creating that number of replicas and spreading them among the replica servers.

## 2.2 Replica placement

Replica placement is the problem of finding good locations to host replicas of a given document. In addition, one needs a mechanism to inform a selected replica server about the creation of the new replica. The most widely used mechanisms are *pull-based caching* and *push replication*. In pull-based caching, when one issues a request for a document to a replica server that currently does not own that document, the replica server fetches it from the origin server. In push replication, the origin server informs the replica server of a replica creation by explicitly pushing the replica to the replica server. Creating a replica of a new document at a replica server is basically the same as creating a replica of an updated document at a replica server. Therefore, we present more details about these inform mechanisms in the next section about consistency enforcement. The remaining of this section discusses in more detail the selection of replica servers that should host replicas of a given Web document.

The replica placement problem consists of selecting $K$ out of $N$ potential replica servers to host replicas of a document, such that some objective function is optimized under a given client access pattern and replica update pattern. The objective function can be minimizing client latency or total bandwidth consumption, or an overall cost function if each link is associated with a cost.

The problem of replica placement can be modeled as the facility location problem or the $K$-median problem [17]. In the facility location problem, given a set of replica servers ("locations") $i$ where one can place a replica ("facility"), placing a replica at location $i$ incurs a cost of $f_i$. Each client $j$ must be assigned to one replica, incurring a cost of $d_j c_{ij}$ where $d_j$ is the demand of the node $j$, and $c_{ij}$ the distance between $i$ and $j$. The objective is to find the number and locations of the replicas such that the total cost is minimized. The $K$-median problem is like the facility location problem. Only there are no costs for placing replicas. Instead, the number of replicas that can be placed is bound to $K$.

However, such solutions can be computationally expensive. This makes it hard to apply them, as placement algorithms are run often. Therefore, simpler solutions are used by existing replica hosting systems.

In [6], *Chen et al.* propose a dynamic replica placement algorithm. The goal is to minimize the number of replicas when meeting clients' latency and servers' capacity constraints. For each document, the replica servers holding a replica are organized into a load-balanced tree with the origin server as root. The algorithm takes as input client requests together with their associated latency constraints. These requests are sent to the origin server, which services the request if the client's latency constraints and the origin server's capacity constraints are met. Otherwise, the algorithm searches for another server in the tree that satisfies both constraints and creates a replica at that server. The algorithm is good in terms of preserving client latency and server capacity constraints. However, it has considerable overhead caused by checking QoS requirements for every client request. In the worst case a single client request may result in creating a new replica. This can significantly increase the request servicing time.

In [11], *Kangasharju et al.* model the replica placement problem as an optimization problem. The goal is to place $K$ objects in some of $N$ servers, in an effort to minimize the average number of inter-AS hops that a request must traverse to be serviced. However, the problem is NP-complete, so finding an optimal solution is not feasible. Therefore, they propose three heuristics. In the first heuristic, each node sorts the objects in decreasing order based on popularity among its clients. Each node stores as many objects in this order as the storage constraint allows. Unfortunately, when placing a replica of a document, the presence of other replicas of this document is not considered. In the second heuristic, a server takes besides the popularity of the document also the server's distance to the origin server into account. Still, replicas at other replica servers are not considered. The third heuristic uses a global replication strategy. For each server/object pair, a cost is calculated based on the document's popularity, the shortest distance between the server and a copy of the object, and the total request rate of the server. Every iteration the server/object pair with the best cost is selected and that object is placed on that server. An iteration ends with recomputing the shortest distances between each server and each document. The algorithm iterates until all the replica servers have been filled. This heuristic outperforms the two other heuristics. However, it has a high computational complexity.

In RaDaR [18], *Rabinovich et al.* run the replica placement algorithm on each replica server. A replica server collects access statistics for all of its documents. The algorithm deletes a replica when its request rate drops below deletion threshold $U$ and it is not the sole replica in the system. If the request rate is above $U$, the algorithm migrates the document to a replica server located closer to clients that issue more than half of the requests, to improve client server proximity. The algorithm calculates the distance using *preference paths*. A preference path is a RaDaR-specific metric that is computed by the servers based on information periodically extracted from the system's routers. Finally, in case migration fails, when the request rate of the document is greater than replication threshold $M$ (with $M > U$), the algorithm creates an additional replica on another server.

In SPREAD [20], replica servers periodically calculate the expected number of requests for every document. If the number of requests exceeds a certain threshold, servers decide to create a local copy of the document. They remove a replica if its popularity decreases. If required, the total number of replicas of a document can be restricted by its owner, by using a hop-counter which gets decremented at every replica server where a replica is created. When the counter reaches zero, no more replicas are placed.

These solutions are not optimal, but still have a large computational cost. Some of them require gathering client access information and computation of request rates for all documents. Furthermore, it can be hard to control the number of replicas. This makes it difficult to determine the degree of fault tolerance.

This thesis proposes a different and simpler solution: place replicas at $K$ randomly selected

servers. This placement algorithm does not take client or network conditions into account when replicas need to be placed. It just selects a random server, possibly considering server constraints like server load or storage capacity.

Placing the replicas at random replica servers makes sense in the case of our system, as the selection of servers that form the replica hosting system is generally done in such a way that clients can be serviced well. Server locations are selected that are good for hosting replicas of many objects. When we place replicas at servers selected in a random manner, it is likely that we place these replicas in several parts of the system. Provided that $K$ is high enough, this makes it unlikely that there are clients that do not have a replica reasonably nearby.

An advantage of this solution is its simplicity. There are no computational costs, we only need a random function. Therefore it is easy and fast. Furthermore, we do not need any gathering of client access information. The most important advantage is that this solution makes decentralization possible. The origin server just has to insert documents and updates and can be assumed unavailable for the rest of the time. The replica servers can handle placing the replicas. A problem that we need to solve when using random placement is locating replicas. We discuss this problem in the next chapter.

## 2.3 Consistency enforcement

Having multiple copies of a document introduces a consistency problem. When one updates a copy of a data item, all the other copies have to be destroyed or brought up-to-date as well to prevent clients from retrieving stale data.

Consistency can be enforced using various *consistency models* depending on the requested consistency requirements. A consistency model dictates the consistency-related properties of documents delivered by the system to its clients. A consistency model can be implemented using various *consistency policies*. A consistency policy defines how, when and to which replicas the various *content distribution mechanisms* are applied. A content distribution mechanism defines how and when replica servers exchange the replica updates.

Section 2.3.1 discusses the various consistency models, where section 2.3.2 gives an overview of content distribution mechanisms.

### 2.3.1 Consistency models

Consistency models dictate the consistency-related properties of documents delivered by the system to its clients. Consistency models differ in how strict they are in enforcing consistency.

*Coherency* is the strongest form of consistency. We consider a system to be coherent when all copies of a document in the system are identical at all times, even in the case of updates. Coherency is not possible in distributed systems. First, in distributed systems one faces transmission delays: it takes some amount of time (unbounded) before updates are disseminated to replica servers. Furthermore, even when one first spreads an update to all replica servers before updating a document, it is very hard to synchronize these servers in order to apply the update at the same time.

It is therefore impossible to define consistency with respect to the state of the application. Consistency models thus define properties in the perspective of clients. Strong consistency means that when clients request documents the system behaves as if it was coherent. This does not mean that the system is coherent: a replica that is different from others, for example because the copy at the origin server is updated, must simply be destroyed or otherwise not be delivered to clients until it is brought up-to-date. Strong consistency is seldom used in wide-area systems due to high synchronization costs.

However, in many cases, strong consistency is not required. When one relaxes on the consistency strictness, *weak consistency* can be applied. Weak consistency ensures that eventually all

updates reach all replicas, possibly bounded by some time or order constraints. As weak consistency is resistant to delays in update propagation and causes less synchronization overhead, it fits better in wide-area systems.

Consistency models usually define consistency along three different axes: time, value and order [30]. Order-based consistency models are generally used in replicated databases. In these models, replicas can differ only in the order of execution of write operations according to certain constraints, e.g. a maximum number of out-of-order operations. However, they need to to timestamp and exchange operations among all replicas. These models are mostly useful to support concurrent distributed updates.

Value-based consistency models [1] are based on the assumption that each replica has an associated numerical value that represents its current content. They define consistency as the numerical difference between two replicas. These models ensure that the difference between the value of a replica and that of other replicas is no greater than a certain threshold. Value-based consistency models can be applied to documents with a precise definition of value, e.g. a Web document containing the current stock rates. Unfortunately, for regular Web documents it can be hard to obtain such a value.

Time-based consistency models [25] define consistency based on real time. These models require a content distribution mechanism to ensure that an update to a replica is visible to the other replicas and clients after a maximum acceptable threshold of time. An advantage of this kind of models is that they are applicable to all kinds of documents and independent of the document's semantics. Alex [5] is a global file system that uses a time-based consistency model for maintaining consistency of FTP caches. The consistency policy in this system guarantees that the only updates that might not yet be reflected on a replica server are the ones that have happened in the last 10% of the reported age of the file.

The goal of this thesis is to propose a replication policy that implements best-effort weak consistency. It adopts a time-based consistency model and ensures that updates are spread in a reasonable limited time interval after an update takes place at the original document. During this time interval the updated document is available for clients, although clients may access stale data. Therefore, we strive to keep the update dissemination time as low as possible.

### 2.3.2 Content distribution mechanisms

Content distribution mechanisms define the exchange of replica updates. They differ on two aspects: the form of the update and the direction in which updates are triggered. The decision for these two aspects is a tradeoff between the degree of consistency one wants to achieve and the communication overhead it introduces.

Replica updates can be transferred in three different forms: *state shipping*, *delta shipping* and *function shipping*. State shipping is the simplest form, just the whole replica is sent. When one uses delta shipping, only differences with the previous version are transmitted. With function shipping, only the operations that cause the changes are sent.

The advantage of delta and function shipping is that they can incur less communication overhead compared to state shipping, as only the actual changes are sent, respectively the size of the description of the operations is usually independent from the object state and size. However, it requires each replica server to have the previous replica version available. Furthermore, delta shipping assumes that the differences between two document versions can be quickly computed, which can be become difficult for regular documents. Finally, function shipping forces the replica servers to be able to perform a, possible computationally demanding, operation.

Although state shipping can incur significant communication overhead, especially when a small update is performed on a large document, there are no complex computations needed. Even more important, it does not require replica servers to have the previous replica version available. This makes the replication policy more flexible as updates can be placed on replica

servers that do not have a copy of the document yet without generating communication overhead caused by retrieving the previous version. This is an important property in order to make the replication policy decentralized as we explain in the next chapter.

The update transfer can be initiated by the replica server that needs a new version, a *pull*, or by the replica server that holds the new replica version, a *push*. Furthermore a combination of both mechanisms can be used.

With a pull-based approach, the replica server determines when to fetch a new version of a document from its origin server. This has the advantage that origin servers do not have to store state information, which leads to higher fault tolerance. However, replica servers have to estimate when to pull an update. A replica server can check on every client request for a document if the origin server has a newer version. If so, the origin server returns the new version, else just a header stating the replica is still up-to-date is returned. This approach has the advantage that strong consistency is possible. Unfortunately, it can incur large communication overhead as the origin server has to be contacted for each request even if the replica is still valid.

In another approach, the replica server computes a *Time To Refresh* (TTR) attribute for each of its documents, which denotes the next time the document should be validated. The value of TTR can be a constant, or can be calculated from the update rate of the document. It may also depend on the system's consistency requirements. Rapidly changing documents or stringent consistency requirements require a small TTR, whereas documents with infrequent changes or less stringent consistency requirements can have a larger TTR. However, enforcing stricter consistency depends on careful estimation of TTR: small TTR values provide good consistency, but at the cost of unnecessary transfers when the document was not updated.

One can also combine both approaches: only after the TTR value expires, validity checking is performed. Although this incurs less communication overhead, strong consistency is not possible and good estimation of the TTR attribute is still important.

The push-based scheme ensures there is only communication when there is an update. This way, one can provide strong consistency without introducing the communication overhead from the validity checking approach: since the origin server is aware of changes, it can precisely determine which changes to push and when. However, the replica server initiating the update transfer needs to keep track of all replica servers to be informed, but it has been shown that storing this list can be done in an efficient way [4]. A more important problem is that the origin server becomes a single point of failure, as the failure of this server affects the system's consistency until it is recovered.

A disadvantage of the push approach is that the origin server has to keep track of all replica servers to be informed. This thesis proposes a replication policy where the origin server can just insert a new (version of a) document into the system and disconnect from the system. The origin server does not have to store state information. It only has to know the addresses of some arbitrary replica servers in the system to which it can send the documents and updates.

In [3], *Bhide et al.* propose a couple of schemes where push and pull are combined. In one scheme they use push and pull simultaneously to achieve advantages of both approaches. Another scheme adaptively chooses between push and pull depending on the data change rate, client requirements or resource availability. A more suitable way of combining push and pull for our replication policy is to allow the former to trigger the latter. This can be done using *invalidations*. A document's origin server pushes invalidations to a replica server. They inform the replica server that the replica it holds is outdated. Then, the replica server can decide to pull the new version from the origin server. In the proposed replication policy, the origin server can insert the invalidation into the system. The replica servers take care of disseminating the invalidation. A replica server that needs the new version can pull it from the origin server, but also from replica servers that have obtained a copy. In this approach the origin server can disconnect after the update is fetched at least once.

## 2.4 Request routing

When a client issues a request for a certain document, the request needs to be routed to a copy of the requested document. To accomplish this, one first has to select a server that handles the request. There are two possible server selection approaches: the server is selected out of (a subset of) the $K$ servers that possess a replica of the requested document, or the server is selected out of (a subset of) all the $N$ servers in the replica hosting system. In the former approach, one has to determine which (subset of the) $K$ servers have a copy of the requested document before selecting one of them. This requires that one has to identify at least one server that has a copy of the requested document. This approach can be useful when identifying replica servers with a certain document copy is easy. In the latter approach, one can select a server out of (a subset of) all the servers in the system. With this approach no identification of servers possessing a copy of the requested document is needed. However, when the selected server does not have a local copy, this server has to find one. This approach may be useful when identifying replica servers with a certain document copy is hard. However, one should be able to locate replicas in an efficient way. We discuss locating replicas in the next chapter.

After the selection approach is chosen, one has to select one (or more for fault tolerance reasons) server that should handle the client request. There are two possible policies for selecting the server: *non-adaptive* and *adaptive*. In non-adaptive policies, current system conditions are not considered when selecting a server. Instead, they exploit heuristics based on assumptions of system conditions. The advantage of these policies is that they are easy to implement. A disadvantage is that these policies only work when the assumptions made by the heuristics are met.

In adaptive policies, current system conditions, e.g. server load, client-server distance and end-to-end latency, are considered when selecting a server. The advantage is that they are able to adapt their behavior to changing situations. However, they achieve this at the cost of a higher complexity. Selecting a server selection policy is a task of the replica hosting system. When system conditions are monitored, the system can decide to use these when selecting a server for handling a client request.

Finally, when a server is selected, one has to inform the client about the selected server to which it is redirected. This can be done using *non-transparent*, *transparent* or *combined mechanisms*. Non-transparent mechanisms reveal the redirection to the clients. They can be implemented with HTTP. An advantage is that these mechanisms are easy to implement. However, they introduce an explicit binding between a client and a given replica server. Transparent mechanisms perform client request redirection in a transparent manner. These mechanisms can be based on DNS. They do not introduce explicit bounds between clients and replica servers, even if the clients store references to replicas. However, they have poor client identification and coarse redirection granularity. One can also combine transparent and non-transparent mechanisms to achieve better results. Like the server selection policy, the inform mechanism is a task of the replica hosting system.

## 2.5 Availability

In a distributed system failures can occur. Replicas can become unavailable because of failing replica servers, or become unreachable because of network failures. Systems such as Akamai assume that a significant and constantly changing number of components or other failures occur at all times in the network [1]. Consequently, the system must be designed such that Web content can be delivered successfully under these circumstances. An important principle Akamai uses is redundancy, especially for DNS servers which direct end users to Web servers. It achieves redundancy in DNS by introducing a two-layer approach combined with returning multiple IP addresses. Furthermore, Akamai continuously monitors the state of services, and their servers

and networks. This way it can detect failures quickly and make sure they do not affect clients, e.g. by directing clients to other, correctly working servers.

In Globule [16] the origin server contains the authoritative version of all documents of its site and normally should be reachable by other servers at all times. However, as the origin server cannot be assumed to be always available, Globule uses one or more backup servers to guarantee availability of the site. These backup servers maintain a full up-to-date copy of the hosted site. When the origin is unavailable, one available backup server is sufficient for the site to work correctly. The goal of replica servers is not increasing availability but maximizing performance. In the worst case a failing replica leads to increased response times. Even when all replica servers fail, the requested document can be found at the origin or one of the backup servers. Globule monitors the availability of origin, backup and replica servers to avoid directing a client to an unavailable server.

Decrepol does not need a backup server for availability. Instead, it spreads all documents of the origin server across the replica servers. When a replica server does not have a local copy of a document it knows where to fetch it. Failing replica servers lead to decreased performance. As long as at least one copy of each document of the site remains available the site is available. Therefore, the replication degree of the documents dictates the degree of availability. In case the origin server fails, we cannot insert new documents or updates into the system, but it does not affect fetching documents currently in the system.

In Decrepol, when a replica server recovers from a failure one can distinguish two cases: the server has no state or it does have a state. In the first case, the replica server is the same as a joining server. It just has to announce its presence and ask for replicas. In the other case, the replica server may contain stale documents as it may have missed some updates during the period it was unavailable. Therefore, it should make sure it fetches the missing updates and brings its documents up-to-date. We discuss these issues at length in Section 2.3.

## 2.6 Replication policies in Globule

In Globule [16] the origin server contains the authoritative version of all documents of its site, and is responsible for distributing contents among other involved servers. Backup servers maintain a full up-to-date copy of the hosted site and guarantee the availability of the site. Replica servers contain only a partial copy of the site and their goal is to maximize performance.

Globule supports multiple replication policies. With *proxy* there is no creation of replicas at all. All replica servers forward their requests directly to the origin server. With *ReplNoCons* the origin server creates replicas at replica servers and keeps it there forever without checking. This policy is obviously useful only in a few extremely specific cases. *TTL* allows a replica server to deliver a copy to a requester without any consistency check during a fixed amount of time $t$ since a fresh copy has been fetched. If the period has expired a consistency check is required. *Alex* is basically the same as TTL, but uses a variable amount of time $a$ after which a consistency check is required. The amount of time can be varied according to for example the update rate of a document, as highly updated documents require a shorter validation time.

TTL and Alex allow a bounded inconsistency whereas our goal is to minimize it. When applying *Invalidation* for a certain document, replica servers owning a copy of this document, register at the origin server. The origin server sends a message to all registered replica servers when the document is updated so that they drop their outdated copy. This however creates problems if certain replica servers are unavailable at the time of a document update. When a replica server that is registered at the origin server becomes unavailable, the origin server cannot send update messages to the replica server. However, when the replica server recovers it needs the update messages it missed during the period it was down in order to update its documents. As the origin server is responsible for delivering these update messages, it has to buffer the

replica server's update messages until the replica server becomes available again.

When the origin server is down, no updates can be inserted to the system. Furthermore, when a replica server needs a document it does not have locally, the replica server needs to contact the backup server for it. The replica server cannot register at the origin server for this document until the origin is available again. The backup server usually delivers the document using TTL. Another option is to let the replica server register at the backup server. However, the backup server needs to forward all its registrations to the origin server when it becomes available again.

An important feature of Decrepol that the previous replication policies do not support is that it allows the origin server to be unavailable most of the time. Furthermore, the replica servers organize themselves such that there is at least one copy of each document among them, possibly more for fault tolerance or performance reasons. Replica servers that do not have a local copy of a document know where they can fetch it. The main advantage of this approach is that the origin server only has to be available when an update needs to be inserted and can be unavailable for the rest of the time. Therefore the origin server can be a laptop or PDA. Furthermore, the origin server is no single point of failure. The replica servers are responsible for spreading replicas and handling client requests in a distributed manner.

# Chapter 3

# Related Work: Peer-to-Peer

This thesis proposes a decentralized replication policy where the replica servers need to organize themselves such that the right number of replicas of a document is placed on the right replica servers, they know where to fetch documents not possessed locally, and the replicas are kept consistent. We only need the presence of the origin server when we have to insert a new document or an updated version into the system.

These properties come close to those offered by peer-to-peer overlays: sharing computer resources, decentralization, self-organizing, resilience to network and server failures. We therefore decided to structure our replication policy following peer-to-peer architectures. Section 3.1 gives an overview of peer-to-peer systems. Section 3.2 discusses unstructured peer-to-peer systems in detail, thereby stating which of their properties can be used for our replication policy and which issues they might introduce. Section 3.3 does the same for structured peer-to-peer systems.

## 3.1 Overview

Peer-to-peer systems are distributed systems with two defining characteristics: sharing of computer resources (e.g. content, CPU cycles, storage and bandwidth) by direct exchange, rather than requiring the intermediation of a centralized server, and the ability to treat instability and variable connectivity as the norm, automatically adapting to failures in both network connections and computers, as well as to a transient population of nodes [2]. One can distinguish three kinds of peer-to-peer systems.

The first is designed to efficiently support *content-based searching*. These systems often use a centralized index server to facilitate interaction between peers by performing lookups and identifying the nodes storing the files. An example of such a *hybrid decentralized* system is Napster [14]. The advantage of such systems is that they are easy to implement and locate files quickly and efficiently. However, they also introduce a single point of failure and therefore are not scalable. This is something we want to avoid in the proposed replication policy, so we need a more decentralized solution. Other peer-to-peer content-based searching systems use supernodes as index servers. These nodes are dynamically assigned and assume a more important role acting as local central indexes for files shared by local peers. Examples of this kind of *partially centralized* systems include KaZaa [12] and Gnutella [8]. Supernodes may be interesting as the network becomes large and can introduce a hierarchical structure to make the system scalable. However, in this thesis we consider networks of only hundreds, maybe thousands of servers.

The second kind of systems is based on *epidemic protocols*. The goal of these systems is the rapid and efficient dissemination of information. A crucial element in an epidemic protocol is that a participating peer can randomly select another peer to exchange information with. Example systems include Newscast [10, 27] and CYCLON [26]. These kinds of systems are

*purely decentralized*, i.e. all nodes in the network perform exactly the same tasks, acting both as servers and clients, and there is no central coordination of their activities. The placement of content is unrelated to the overlay topology. An advantage of these networks is that they are easily maintained. However, as there is no correlation between content and nodes, content typically needs to be located. This can be done by brute force methods such as flooding, but such a solution can cause availability and scalability problems. Therefore a more sophisticated location method is needed. We discuss peer-to-peer systems based on epidemic protocols in Section 3.2.

The third kind uses a structured overlay network, i.e. the network formed on top of and independently from the underlying physical computer network, for efficiently routing a request to its destination. The overlay topology is tightly controlled and files are placed at precisely specified locations. Examples of such systems include Chord [23], CAN [19], and Pastry [21]. All structured systems are inherently purely decentralized as form follows function. An advantage of structured systems is that they provide a scalable solution where queries can be efficiently routed to the node with the desired content. On the other hand it can be hard to maintain the structure required for efficiently routing messages in the face of a very transient node population, in which nodes are joining and leaving at a high rate. We discuss structured peer-to-peer systems in Section 3.3.

## 3.2 Unstructured peer-to-peer systems based on epidemic protocols

There are many epidemic protocols, but we discuss only a few of them. Our main focus is on Newscast and CYCLON, as they form the basis of our full replication policy and of the unstructured version of our partial replication policy. In addition we discuss some protocols based on CYCLON, namely CYCLON-VICINITY, T-Man and Sub-2-Sub, as they offer solutions for locating replicas that can be used in our partial replication policy.

The Newscast protocol [10, 27] is an epidemic protocol that combines information dissemination with efficient membership management in large, dynamically changing sets of autonomous agents. Agents can join and leave at virtually no cost at all, and without affecting the information-dissemination properties of the protocol.

Newscast is based on a collection of agents where each agent can provide news. Every agent has an associated correspondent running on the same machine hosting the agent. All correspondents together form a news agency responsible for spreading the news to all agents. Figure 3.1 shows the overall architecture of Newscast. The definition of what counts as news is application dependent. Each correspondent maintains a cache of at most $c$ news items. When a correspondent receives a news item from its agent, the former timestamps the item, adds its network address to it, and subsequently adds the item to its cache. A news item consists of an agent identifier and the actual news the agent provides. Correspondents periodically exchange caches. This is done as follows. The correspondent asks for a fresh news item from its agent and adds it to its cache. It selects randomly a correspondent by considering the network addresses of other correspondents in the cache. The cache contains only a small subset of network addresses of peers in the system. The correspondent sends all items in its cache and receives all cache items from the other correspondent. After the exchange both correspondents pass the received items on to their agent and add them also to their cache. Subsequently the correspondents keep the $c$ freshest items in their cache according to the timestamps.

An advantage of Newscast is the simple membership management protocol. When a node wants to join the system, it just has to identify the network address of any arbitrarily chosen correspondent and initialize its own cache with the one of that correspondent. Leaving the system requires no action at all. A correspondent will be forgotten in a limited amount of time

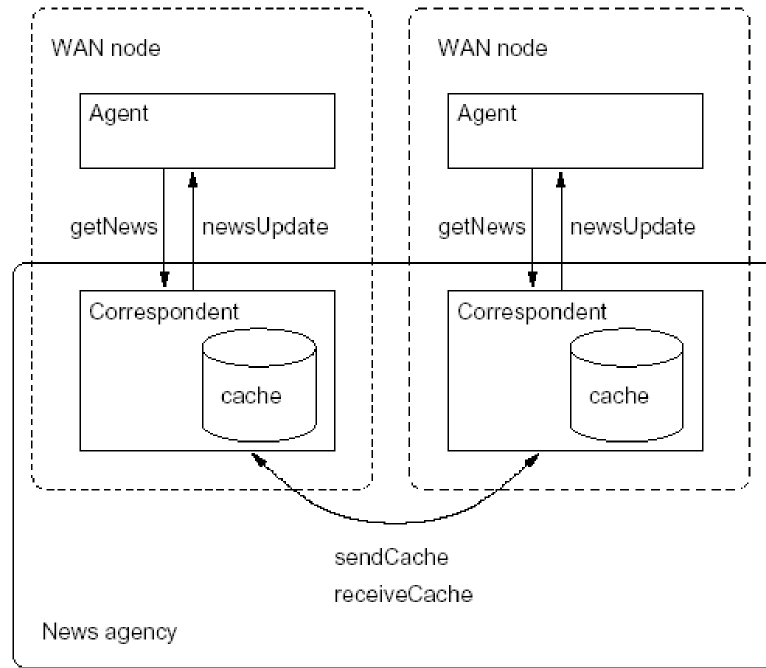


Figure 3.1: The organization of a Newscast application.

when it does not provide news items anymore. Therefore, Newscast is robust to node failures. Another important advantage of Newscast is that it can disseminate information quickly to all nodes in the system. Furthermore, there is no global synchronization required among all correspondents. They just need to keep time consistent within a single cache, which can be achieved by exchanging local time and modify timestamps of received items accordingly.

Newscast considers the latest news from an agent most important and the fresher the news is the better. Old information is thrown away whenever a fresher item from the same source is available. However, in a Web hosting system things are a little different. What is considered most important is not the latest news item of an origin server (e.g. announce of a new document or update), but the latest version of a document. This means that if we announce each new version of a document in a news item, it should be possible for two news item of the same source to be in a correspondent's cache at the same time, otherwise, an origin server could only insert a new item when the previously inserted news item has reached all necessary replica servers in the system and can be removed from the caches.

Another problem when directly mapping a Web replication policy on Newscast might be locating content. Newscast is about disseminating news to all nodes in the network. We can consider announcing a new document or update as news. When we place this document on all nodes in the network, as is the case with full replication, locating a copy is not an issue as one can just ask an arbitrary node. However, in the case of partial replication only a subset of the nodes in the network should store a copy of the document. When one requests a partially replicated document from a server that does not have a copy stored locally, the server needs to locate a copy by searching the system. Although Newscast tends to lead to small average path lengths [27], when there is no correlation between documents and nodes, and the number of replicas is relatively small compared to the number of nodes in the network, searching can still become a complex task taking a lot of time and/or requiring a lot of communication and thus bandwidth.

Another important disadvantage of Newscast is that it creates communication graphs with a relative high clustering coefficient [27], which, for a given node, is the fraction of pairs of its neighbors that are also neighbors of each other. This does not only weaken the connectivity of

a cluster to the rest of the network, and therefore increasing the chances of partitioning, but is also not optimal for information dissemination due to the high number of redundant message deliveries within highly clustered parts of the network [26].

CYCLON [26] is a gossip-based protocol that looks very much like Newscast. However, CYCLON differs in a few aspects from newscast. Firstly, nodes exchange only subsets of their caches instead of whole caches. This requires less bandwidth without affecting the connectivity of the overlay. Secondly, a node selects the node with the oldest timestamp in its cache to gossip with instead of selecting a peer randomly. This way the time a pointer can be passed around until it is chosen by some node for gossiping is limited, resulting in a more up to date overlay. Furthermore, selecting the node with the oldest timestamp leads to a predictable lifetime of each pointer and controls the number of existing pointers to a given node at any time. Finally, there is a difference in deciding which items to keep after gossiping. CYCLON replaces, in case the cache is full, sent items by received items, whereas Newscast nodes keep the $c$ freshest items.

A disadvantage of CYCLON compared to Newscast is that the former requires a more complex join operation. When a new node $P$ wants to join the network, like in Newscast it has to know any single node that is already part of the network, which we refer to as *introducer*. However, instead of just initialize the cache of node $P$ with the cache of the introducer, the introducer initiates cache size $c$ random walks of length at least equal to the average path length. A node $Q$ where a random walk ends replaces a random entry of its cache with a fresh entry of node $P$. Furthermore, node $P$ adds the replaced item of node $Q$ to its cache. This way, the cache of node $P$ is not only filled with randomly chosen nodes in the network, but also the number of references to $P$ is equal to the cache size, and the number of references to the other nodes in the network has not been modified. Random walks may fail because of node failures or an unreliable network. Fortunately, a node can join by being involved in an exchange with a single other node, although it takes some more rounds before its cache is full [26].

The main advantage of CYCLON is that it creates communication graphs with a lower clustering coefficient than newscast does. The coefficient is practical equal to the clustering coefficient of randomly created graphs. However, CYCLON does not solve the content location problem.

In [29], *Voulgaris et al.* propose a two-layered approach to allow for searching based on grouping semantically related nodes, as shown in Figure 3.2. The bottom layer is CYCLON. CYCLON creates an overlay with completely random, uncorrelated links between nodes, and feeds the top layer with random nodes to make sure the top layer adapts to changes in the network, e.g. joining nodes and nodes with changed content. The top layer, VICINITY, is dedicated to grouping semantically related nodes. Each node has a semantic view, which is a dynamic list of semantic neighbors. A node first queries its semantically close peers before using search methods that span the entire network. The goal of this system is to organize semantic views so as to maximize the hit ratio of the first phase of the search. This approach is very suitable for content-based searching. However, in a Web hosting system nodes generally do not have such a semantic relation.

T-Man [9] is a protocol that can be used in large distributed systems for constructing and maintaining different types of topologies in a gossip-based fashion. It uses a ranking function that defines the topology by ranking nodes according to increasing distance from any given node. The goal of T-Man is constructing and maintaining a target topology by connecting all nodes in the network to the right neighbors according to the ranking function.

T-Man differs from Newscast and CYCLON in two ways. First, it selects a node for gossiping by first applying the ranking function to order the items in the view and then select a random item from the first half of the view instead of selecting from the view a completely random item or the oldest item respectively. Second, after a node has exchanged items, it uses the ranking function to order the items in the view, and instead of keeping the $c$ freshest items, it keeps the first $c$ items according to the ranking function. Furthermore, T-Man feeds the protocol with

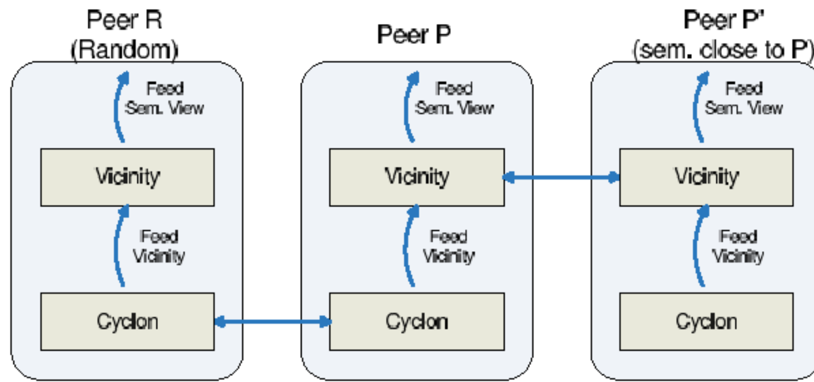


Figure 3.2: The two-layered protocol.

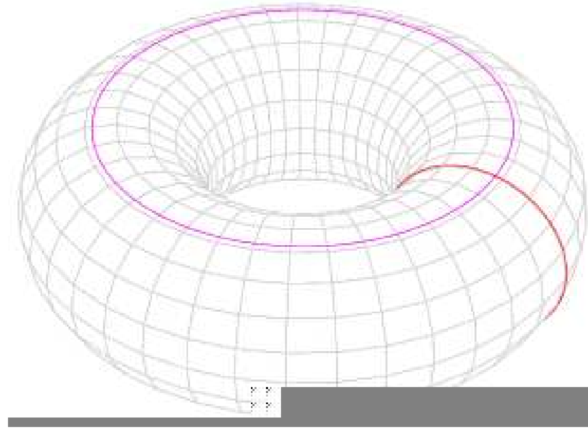

Figure 3.3: A torus.

random nodes in the network using a service like CYCLON, in order to adapt to changes in the network.

Using T-Man, we can construct the servers in a Web hosting system in a topology that facilitates locating documents. We can create such a topology by assigning the servers a server identifier, for example by hashing the server's address, and selecting a certain ranking function. We can also assign the documents an identifier, for example by hashing the document's URL. This way, we can create a correlation between servers and documents, where we place a document at a server with an equal identifier or an identifier that is close to that of the document. When one requests a document from a server that does not have the document stored locally, the server knows it has to find a server with an identifier closer to that of the requested document. The requested server can find such a server when we have constructed a suitable topology.

A suitable topology for searching purposes has a low diameter and includes hints on how to get closer to the destination. A torus is an example of such a topology. It is a three-dimensional figure that is the product of two circles as shown in Figure 3.3. One can create it from a mesh where the opposite edges are connected in both horizontal as vertical direction.

Sub-2-Sub [28] is a content-based publish/subscribe system. In such a system subscribers express their interest in data by registering subscriptions. Subscriptions are represented by arbitrary predicates on attributes. Publishers notify subscribers of events matching their subscriptions.

Sub-2-Sub uses VICINITY to automatically cluster subscribers with similar interests. Once subscriptions are clustered, Sub-2-Sub sends events directly to the matching cluster where they are efficiently disseminated. It uses CYCLON to discover new nodes and to keep the overlay connected in a single partition. Finally, to make sure all interested subscribers receive an event,

it organizes subscribers that have the same interest in bidirectional rings.

The main difference with Web hosting systems is that in publish/subscribe systems servers where data needs to be routed to, express their interest in the data. In Web hosting systems, replica servers do not express their interest, instead the goal is to find a suitable set of replica servers to store the replicas of a document. However, when we can distinguish a server $P$ where a replica needs to be placed, by using identifiers for both documents and servers as explained before, and a suitable topology is created using T-Man, then like in Sub-2-Sub Vicinity can be used to spread the replicas across servers with an identifier close the identifier of node $P$.

However, when we use identifiers for both documents and servers as explained before, we can distinguish a server $P$ where we need to place a replica. Furthermore, when we create a suitable topology using T-man, we can locate this server. Finally, we can use Vicinity to spread the replicas across servers with an identifier close to the identifier of node $P$.

## 3.3 Structured peer-to-peer systems

There are many structured peer-to-peer systems acting as distributed hash tables. We discuss only a few of them. Our particular attention is focused on Chord, as it forms the basis of our structured partial replication policy, and CFS, which is a storage system based on Chord. In addition we briefly discuss CAN and Pastry and point out the main differences of these systems with Chord.

Chord [23] is a peer-to-peer lookup protocol. It acts as a distributed hash function mapping keys onto nodes. Such a node might be responsible for storing a value associated with the key. Chord uses *consistent hashing* [13] to assign keys to nodes. An advantage of consistent hashing is that it tends to balance load as each node is assigned roughly the same number of keys. Another advantage is that the number of keys that are reassigned to other nodes when a node joins or leaves the system is small: a $O(1/N)$ fraction of all the keys in the system is reassigned when an $N$th node joins or leaves the system.

In Chord, the routing table is distributed, so each node needs to store information about only a few other nodes, namely $O(logN)$. That information is needed for efficient routing, but performance degrades gracefully when that information is out of date. In the worst case, only one piece of information per node needs to be correct in order to guarantee correct routing of queries. Chord automatically adjusts its internal tables to reflect newly joined nodes as well as node failures.

The consistent hash function assigns each node and each key an $m$-bit *identifier*. A node's identifier is chosen by hashing the node's IP address, whereas a key identifier is produced by hashing the key. Identifiers are ordered on an identifier circle modulo $2^m$. Key $k$ is assigned to the first node whose identifier is equal to or follows the identifier of $k$ in the identifier space. This node is called the *successor* node of key $k$. Figure 3.4 shows an identifier circle with $m = 6$, consisting of ten nodes storing five keys. In Chord, each node maintains a routing table with up to $m$ entries, called the *finger table*. Each node stores information about only a small number of other nodes, and knows more about nodes closely following it on the identifier circle than about nodes farther away. More precisely, each node has finger table entries at power of two intervals around the identifier circle. Figure 3.5 shows the finger table entries for node 8. A node's finger table generally does not contain enough information to directly determine the successor of an arbitrary key $k$. Therefore, a node needs to communicate with other nodes in order to perform a lookup.

A lookup of key identifier $id$ by node $n$ consists of finding the successor node of $id$. If $id$ falls between $n$ and its successor, $n$'s successor is also the successor of $id$. Otherwise, node $n$ asks the node $p$ in its finger table whose identifier immediately precedes $id$ to find the successor node of $id$. The reason for this choice is that the closer $p$ is to $id$ the more it will know about

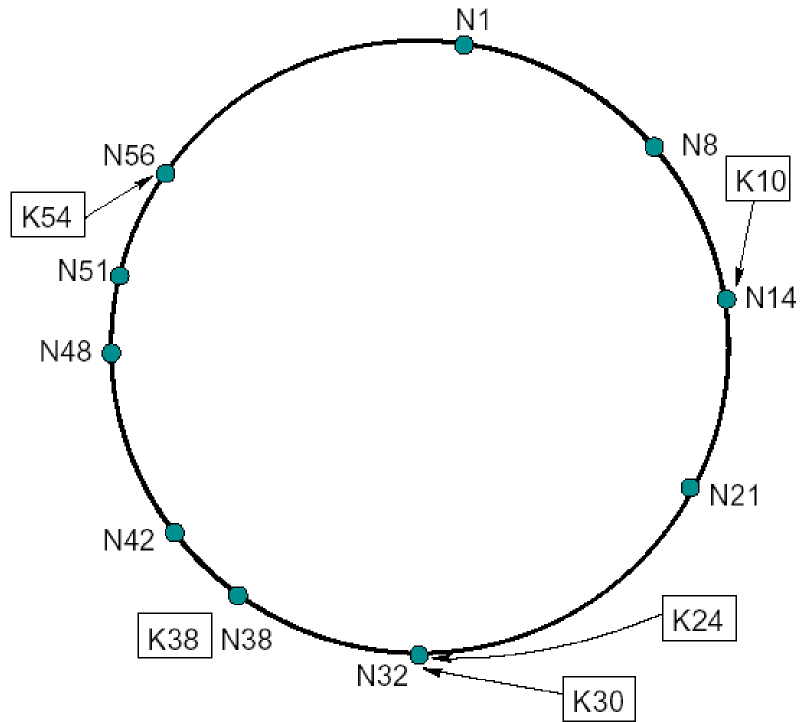


Figure 3.4: An identifier circle consisting of ten nodes storing five keys.

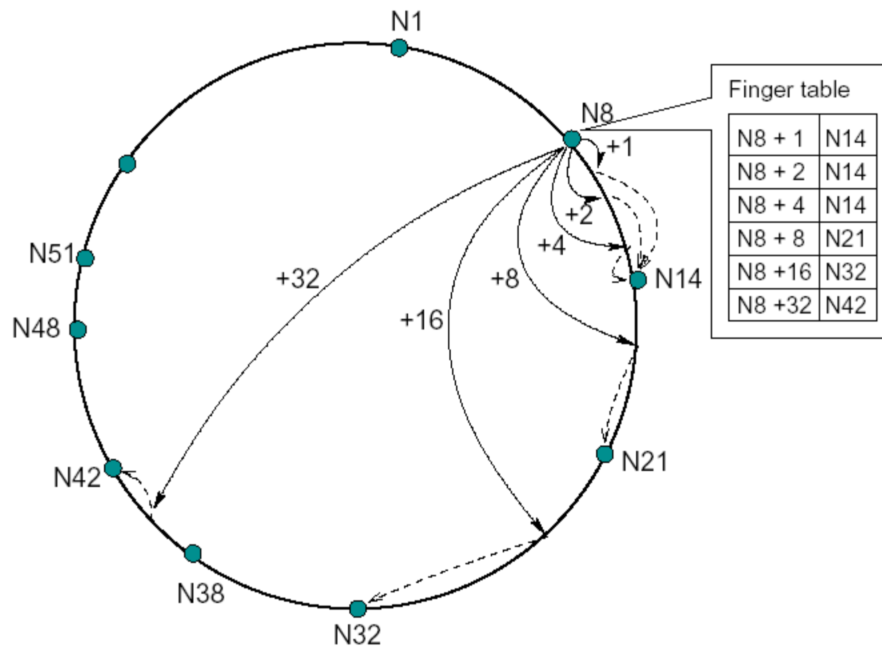


Figure 3.5: The finger table entries for node 8.

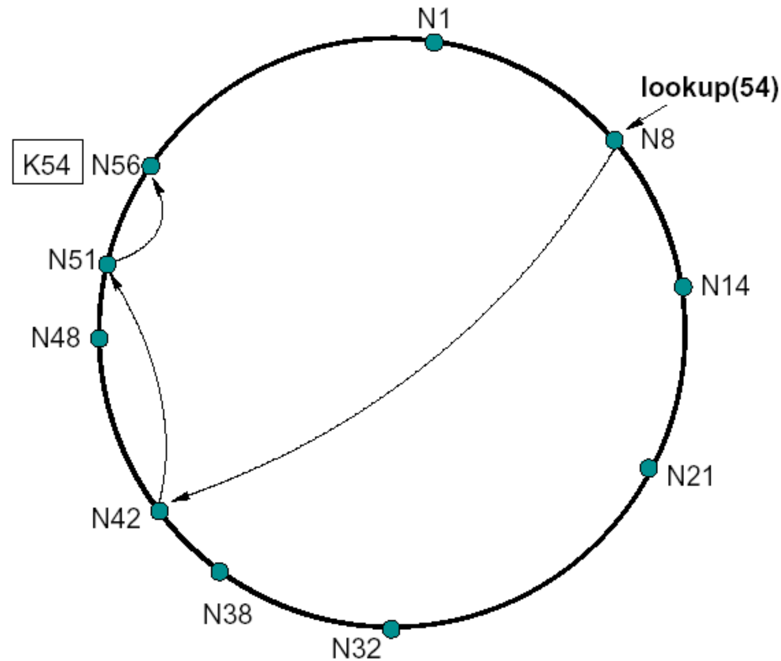


Figure 3.6: The lookup of key 54 starting at node 8.

the identifier circle in the region of $id$. Since each node has finger table entries at power of two intervals around the identifier circle, each node can forward a lookup query at least halfway along the remaining distance between the node and the target identifier. A lookup requires $O(logN)$ messages in an $N$-node network. Figure 3.6 shows the lookup of key 54 starting at node 8.

When we map our replication policy on Chord, we need to associate a key with each Web document, for example by hashing its URL. Then we can place a document on the successor node of its key. In Chord, nodes can maintain a successor list containing the node's first $r$ successors in order to increase robustness. If a node's immediate successor does not respond, the node can contact the next successor in the list. We can use this successor list mechanism to store multiple replicas of a document in the system, by storing replicas of a document on the first $k$ successor nodes of the associated key.

When a node $n$ joins the network it asks an arbitrary node in the network to find the successor of its identifier, and stores this information in its routing table. Furthermore, certain keys previously assigned to $n$'s successor now become assigned to $n$. For our replication policy this means that we need to transfer the documents associated with these keys to their new successor node. A stabilization protocol that runs periodically updates finger table and successor entries to ensure lookups execute correctly as the set of participating nodes changes. When node $n$ leaves the network, all of its assigned keys are reassigned to $n$'s successor. This means for our replication policy, that when a node is about to leave, it can transfer its documents to its successor and notify its predecessor before leaving.

The main advantage of Chord is its efficient location of data items, and the fact that lookup operations run in predictable time and always result success or definitive failure. However, mapping a Web replication policy directly on Chord introduces a number of issues that need to be solved. The first issue is maintaining consistency of the replicas, especially in the presence of failures. Secondly, Chord does not support load balancing between replicas in the successor list. Currently, it always contacts the first successor. In case this successor is unavailable, it selects the next one. The last issue is routing table maintenance in the presence of joining and failing server, and its effect on lookups.

The Cooperative File System (CFS) [7] is a peer-to-peer read-only storage system based on Chord. A CFS file system is read-only from the client's perspective, but the owner of a file system can update it. CFS stores file blocks instead of full documents, so documents can be spread over multiple servers in the system. This prevents large files from causing unbalanced use of storage. CFS uses Chord to locate the servers responsible for a block. This way, it can achieve load balance by spreading the blocks of popular large files over many servers. CFS replicates each block at a small number of servers to provide fault tolerance.

CFS places a block on its successor node. Furthermore, it places replicas of the block at the $k$ servers immediately after this successor node on the Chord ring. The successor node is responsible of making sure that $k$ of the successors in its $r$-entry successor list have a replica of the document. Therefore, CFS must be configured so that $r \geq k$. However, when a document has a high number of replicas, the successor list becomes large.

CAN [19] is also a distributed hash table mapping keys onto nodes. The main difference with Chord is that CAN uses a $d$-dimensional Cartesian coordinate space. Each node maintains $O(d)$ state, namely its immediate neighbors in the coordinate space. The lookup costs are $O(dN^{1/d})$. Thus, the state maintained by a CAN node does not depend on the network size $N$, but lookup costs increase faster than $logN$. Furthermore, CAN requires an additional maintenance protocol to periodically remap the identifier space onto nodes.

In contrast with Chord, Pastry [21] takes the network topology into account to reduce routing latency. However, it achieves this at the cost of a join protocol that initializes the routing table of the new node by using the information from nodes along the path traversed by the join message. As network proximity is out of the scope of this thesis and Chord can be adjusted to include the network topology [7], Chord seems a more suitable candidate for our replication policy than Pastry.

# Chapter 4

# A Full Replication Policy

This thesis presents a decentralized replication policy for Web documents that allows for controlled partial replication. However, such a replication policy is quite complex. Therefore, we first discuss a simpler replication policy that achieves *full replication*, i.e. the replicas of a Web document are spread across *all* replica servers in the system. Full replication is a special case of replication where the number of replicas is equal to the number of replica servers. The main issue for a full replication policy is to make sure all replica servers have an up to date copy of all documents in the system. We can use the solution for this problem as the base of a replication policy that achieves a more general form of replication, which we refer to as *partial replication*. With partial replication, the replicas of a Web document are spread across $k$ replica servers in a network with $N$ nodes and $0 < k \leq N$. The partial replication policy has to deal with a number of additional issues, such as locating a replica in case of a document request to a replica server that does not have the document stored locally, and inserting the exact number of replicas in case the number of replicas is smaller than the number of replica servers. The next chapter introduces a replication policy that achieves partial replication.

The main goal of the full replication policy is to make sure all replica servers have an up to date copy of all documents in the system. As discussed in Section 2.3 about consistency, with up to date we mean that the propagation delay of a document update to all replicas remains within reasonable limits. As also discussed in the consistency section, we accomplish this form of consistency using invalidation. The origin server puts an *insert notification* into the system announcing a new document or a document update. The main challenge for the full replication policy is to spread the insert notification across all replica servers in an efficient and decentralized way. These requirements are very similar to the properties of epidemic protocols discussed in Section 3.2. Therefore, we structure the full replication policy following epidemic protocols as these protocols have the property of rapid and efficient dissemination of information.

Section 4.1 describes the problems we need to solve when we structure our full replication policy following epidemic protocols. Section 4.2 presents the performance evaluation of our policy and Section 4.3 concludes.

## 4.1 The problem

We base our full replication policy on CYCLON [26]. We choose CYCLON over Newscast [10], as the former creates communication graphs with a lower clustering coefficient, which is good for the connectivity of the overlay and decreases the probability of redundant message delivery (as discussed in Section 3.2). CYCLON also takes care of membership management and of keeping the overlay connected by means of gossiping, i.e. all servers periodically exchange subsets of their caches containing server identifiers of other servers in the network, with a server also selected from their caches. We just need to add to CYCLON a way to spread the insert notifications,

similar to the way Newscast spreads news. Using the Newscast method a gossip item would contain an insert notification in addition to the agent identifier of the agent inserting the news and a time stamp. However, Newscast considers the latest news item of an agent most important and older items of the same agent are simply dropped. Thus a cache can contain only one news item per agent at the same time. In a Web hosting system this would mean an origin server can insert only one insert notification at a time. Even worse, we can insert the next notification only after the previous one has reached all replica servers. Certainly, this is not an ideal situation.

A solution to this problem is to decouple the insert notifications from the server identifiers and treat them as separate news items. So, a cache contains server identifier items for membership management and keeping the overlay connected, and insert notification items for spreading the information about a new document or document version. Decoupling also means that the replication policy can use different strategies to spread the two types of news.

Other issues we have to deal with are about the gossiping. The gossip framework consists of seven steps:

1. Select server $P$.
2. Select items to send.
3. Send selected items to $P$.
4. Receive items from $P$.
5. Handle received items (fetching corresponding documents).
6. Add received items to cache.
7. Select items to keep in cache.

One has to select a server to gossip with, decide which items to send and then actually send the chosen items. When the selected server receives the items from the gossip initiator, it also selects items and sends them back to the gossip initiator. After the exchange, both servers handle the received items, which means fetching a document upon receiving an insert notification of a document (version) not yet locally stored. After adding the received items to the cache, the cache may contain more elements than the cache size $c$ allows. The gossip ends with deciding which $c$ items to keep in the cache. The three main elements of spreading the insert notifications are therefore *server selection*, *selecting items to send*, and *selecting items to keep*.

### 4.1.1 Server selection

First, a server has to select another server from its cache to gossip with. CYCLON chooses the oldest server identifier available in the cache. This way, it selects a server that has not been contacted recently, which increases the probability of receiving new information. Another advantage is that server identifiers have a limited lifetime. This means that CYCLON selects server identifiers in a cache after a limited time interval, and server identifiers of failing servers do not remain in caches for an unpredictable time. Another option is to select a server randomly from the cache. However, this way no guarantees can be given over the time it takes before a server in a cache is selected for gossiping.

### 4.1.2 Select items to send

The second element of spreading insert notifications is selecting the items from the cache for gossiping. The gossiping involves two kinds of items: server identifiers (for keeping the network

connected and handling membership management), and insert notifications (to spread information about a new document or update to all replica servers). CYCLON does not exchange all server identifiers in cache. Instead, it selects a subset of $g$ server identifiers. This generates less traffic and thus requires less bandwidth. Furthermore, it is not necessary to exchange all server identifiers in order to keep the overlay network connected. The subset of server identifiers consists of the identifier of the server initiating the gossip and $g-1$ randomly chosen server identifiers from its cache. The other gossip partner selects $g$ random server identifiers.

Selecting all insert notifications from the cache will generate a lot of traffic and thus can require quite some bandwidth. Therefore, we choose only a subset of $g$ insert notifications. A simple solution is to randomly select $g$ insert notifications. This way all insert notifications have an equal probability of being selected. However, older insert notifications may be less interesting as they might have been selected before, or have reached (almost) all replica servers. Therefore it seems worthwhile to choose newer insert notifications over older ones.

However, if one always selects the $g$ newest notifications, it will be harder for notifications to reach all replica servers, because the servers may start selecting newer notifications for gossiping before the older ones have reached all replica servers. On the other hand, if one selects the $g$ oldest notifications, it will be harder for newer notifications to spread, which may lead to an increased dissemination time. A better solution might be to give newer insert notifications a higher probability to be selected than older ones. Section 4.1.4 discusses a couple of probability functions that we can use to achieve this. Furthermore, we compare them against each other and against random selection.

### 4.1.3 Select items to keep

The last element is deciding which items to keep in the cache after a gossip. After a gossip, a server has received some new items (both server identifiers and insert notifications), therefore the number of items in the cache may exceed the cache size $c$. A server has to decide which items to keep in the cache. In CYCLON, a server always removes identifiers pointing to its self and duplicate identifiers. If the cache size is still exceeded, it removes sent server identifiers. So in practice the received items replace the sent ones. This is done in order to control the number of references to servers. Another possibility is to keep the $c$ freshest identifiers in cache.

We can also replace sent insert notifications by received ones. However, this may lead to migrating notifications from server to server instead of replicating, and replication leads to a higher dissemination speed. Another option is to keep the $c$ freshest insert notifications. However, insert notifications may be removed from caches before reaching all replica servers. A better solution might be to give newer notifications a higher probability of being selected to remain in the cache than older notifications. Section 4.1.4 discusses a number of probability functions that we can use for selecting notifications to keep in the cache and we compare them against each other and against random selection.

### 4.1.4 Probability functions

Two important elements of spreading the insert notifications across all replica servers are selecting items to send and selecting items to keep in the cache after a gossip has taken place. We can use a probability function in order to give newer insert notifications a higher probability of being selected for sending. The first function, called $AGE$, is based on the age of the insert notifications, giving younger insert notifications a higher probability of being selected. The probability of a notification $n_x$ to be selected is:

$$P(n_x) = \frac{\frac{1}{age(n_x)}}{\sum_{i=1}^{c} \frac{1}{age(n_i)}} \text{ with } n_x,\, n_i \in \text{updates in cache and cache size } c.$$

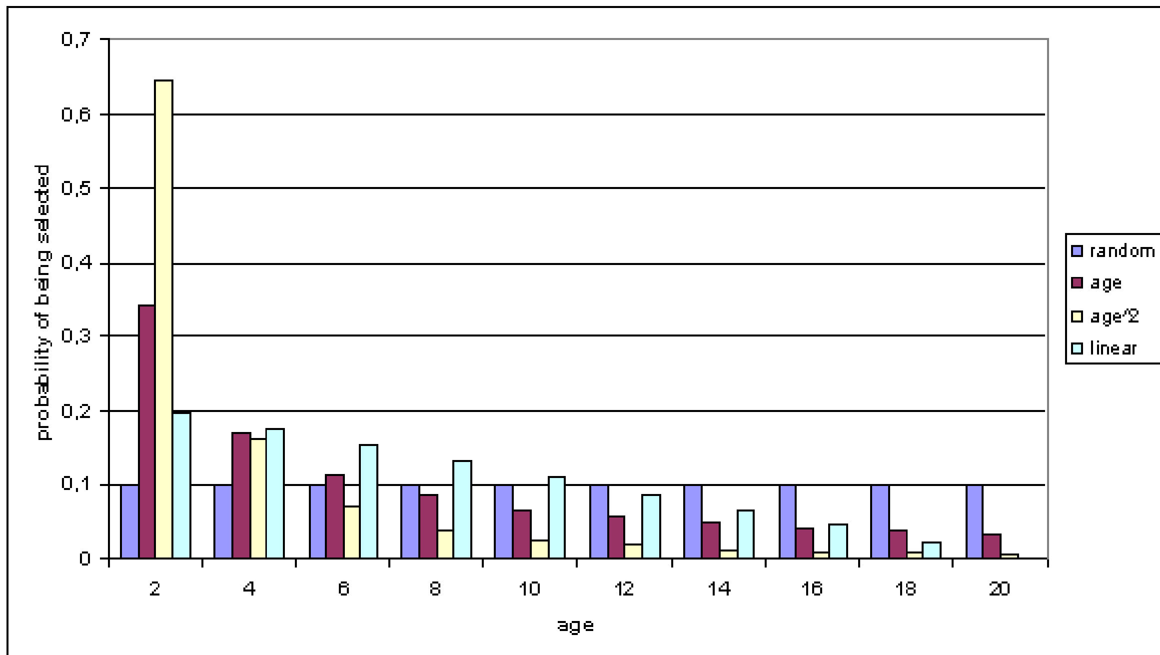


Figure 4.1: Probability functions.

| 2 | 4 | 6 | 8 | 10 | 12 | 14 | 16 | 18 | 20 |
|---|---|---|---|---|---|---|---|---|---|

Figure 4.2: Cache with insert notifications sorted on age.

The second function, *AGE2*, is similar to *AGE*, but it is based on $age^2$ instead of *age*. This gives younger insert notifications an even higher probability of being selected compared to the previous function. The probability of a notification $n_x$ to be selected is:

$$P(n_x) = \frac{\frac{1}{age^2(n_x)}}{\sum_{i=1}^{c} \frac{1}{age^2(n_i)}} \text{ with } n_x,\ n_i \in \text{updates in cache and cache size } c.$$

In the last function, *LINEAR*, the probability for a notification of being selected decreases linearly with the age of the notification. Figure 4.1 shows the probability graphs of the three functions for the cache shown in Figure 4.2. It also shows the probability graph of random selection. When we use random selection, the probability of being selected is equal for all notifications.

We can also use a probability function in order to give newer insert notifications a higher probability of being kept in cache after a gossip. We use the same functions as for selecting items to send.

## 4.2 Performance evaluation

We ran our experiments on a simulator implementing the full replication policy. It simulates 128 replica servers and an origin server. Before we insert any documents into the system, the simulator runs a warm-up phase in order to fill the caches with server identifiers and to create a connected overlay. In the warm-up phase, we fill up the caches with random server identifier of servers in the network. This creates a random overlay network and has the same result as performing the join operation of CYCLON that we discussed in Section 3.2. Furthermore, we fill the insert notification part of the caches by inserting enough warm-up documents to fill up the caches. A server identifier contains an address in order to contact the server, and a timestamp

that indicates its freshness. An insert notification contains the name of the document to be inserted and its last modification date.

We set the initial value of the cache size to 20 for both server identifiers and insert notifications. The *gossip length*, which is the number of items that are selected for gossiping, is set to 3 for both items. We insert a document into the system by adding a corresponding insert notification to the cache of the origin server. When a server needs to fetch the corresponding document of an insert notification, the server fetches it from the gossip partner that sent the insert notification. We assume a replica server can retrieve a document within one gossip round. During a gossip round, each server initiates one gossip. We run all experiments 20 times and we use the median of these experiments as results.

### 4.2.1 Select items to send (infinite cache size)

We first want to determine the best function for selecting insert notifications to send. Therefore, we start with an experiment where we use the four probability functions: *RANDOM*, *LINEAR*, *AGE* and *AGE2*. We do not use a select items to keep function yet and therefore use a cache with an infinite cache size. Only duplicate insert notifications are removed from the cache after a gossip. After the warm up phase, we insert a document and another one 50 rounds later. We are interested in how many rounds it takes before the documents are fully spread across all replica servers. We expect that the three probability functions that give newer insert notifications a higher probability to be selected than older ones, perform better than the *RANDOM* function. First, older notifications in a cache might have been selected for gossiping before and may have reached (almost) all replica servers. Second, new notifications are often selected for gossiping and thus might have been fully spread before newer notifications are inserted into the system.

Figure 4.3 shows the median of how many rounds the two insert notifications take in order to reach all replica servers for the four select to send probability functions. We see that, as we expected, the *RANDOM* function performs worst. It takes 39 rounds for an insert notification to reach all replica servers. The *LINEAR* function performs almost twice as fast with 21 rounds. The results of *AGE* and *AGE2* are almost equal. They outperform the two other functions: an insert notification needs only 5 rounds in order to be fully spread. It therefore seems that functions *AGE* and *AGE2* are the best candidates for selecting items to send when we use caches with an infinite size.

### 4.2.2 Select items to send (finite cache size)

Next, we want to determine the influence of a finite cache size on the number of rounds it takes for replicating documents to all replica servers. We select the simplest select items to keep function *RANDOM*. After the warm up phase, we insert 80 documents, one every five gossip rounds. We also conduct an experiment where we insert a document every two rounds, in order to see whether a higher insert frequency influences the performance of the select items to send functions. We should achieve a lower dissemination speed than with the previous experiment for two reasons. First, caches might drop notifications before they have reached all replicas. Second, there is more competition between notifications in getting selected, especially when we insert notifications every two rounds. Note that relatively frequent insertions are realistic with respect to common access patterns to Web sites, as document updates are quite bursty and can occasionally happen at reduced time intervals.

Figure 4.4 shows the dissemination progress of an insert notification when we insert notifications every five rounds. We see that the *RANDOM* select items to send function clearly performs worst. After 100 rounds, a notification has reached only slightly more than 50 replica servers. The *AGE* and *LINEAR* functions perform better with 23 and 22 rounds respectively. Though, *AGE* has some trouble reaching the last 2 percent of the replica servers. *AGE2* out-

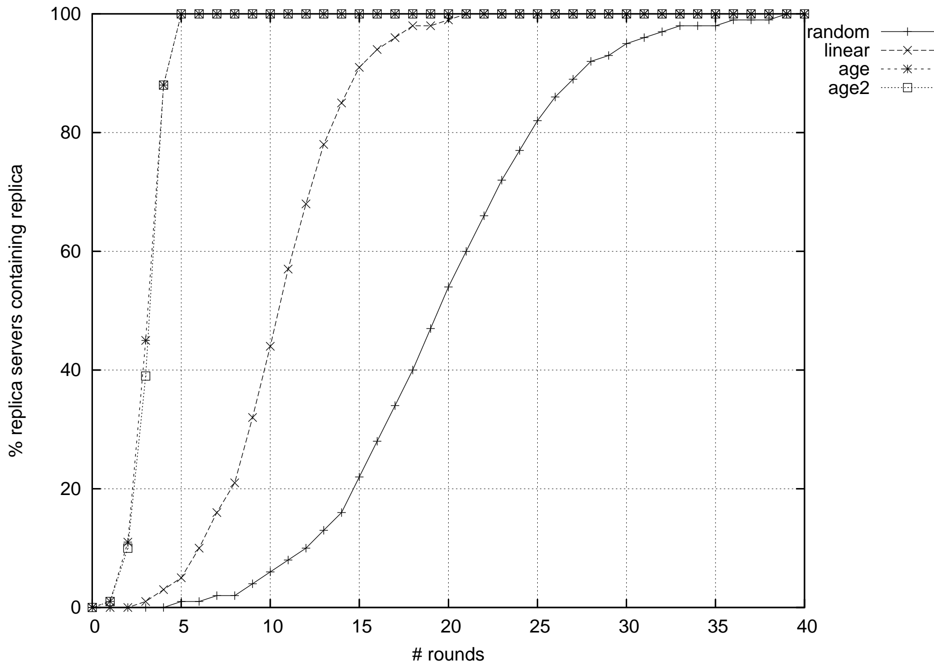


Figure 4.3: Dissemination progress of an insert notification using various select to send functions and an infinite cache size.

performs the other functions. We see that an insert notification only needs 7 gossip rounds in order to reach all replica servers. This indicates *AGE2* is the best candidate for selecting items to send when we use caches with a finite cache size.

However, when we insert notifications every two rounds instead of every five rounds, the dissemination progress of a notification changes dramatically as shown in figure 4.5. We see that when we use *AGE2*, an insert notification still has not reached all replica servers after 100 gossip rounds. *AGE* needs 58 rounds to reach all replica servers. *LINEAR* performs best, as an insert notification only needs 24 gossip rounds in order to reach all replica servers.

When comparing the different functions under both workloads, *RANDOM* obviously performs worst. The two age functions tend to give newer insert notification a too high probability of being selected for gossiping. This way, insert notifications have some trouble reaching all replica servers when there is a high insert frequency of insert notifications. *LINEAR*, on the other hand, does not always provide the best performance but is a safe option as its performance is relatively predictable under various update scenarios. Therefore, we decide to use *LINEAR* for our select items to send function.

### 4.2.3 Select items to keep

Previous experiments show that *LINEAR* is the most suitable function for selecting items to send, so we use this function in the rest of our experiments. The next step is to determine the best select items to keep function. Therefore, we run the previous experiments again, only the select items to send function remains constant and we vary the select items to keep function. We expect that the three functions that give newer notifications a higher probability to stay in the cache after a gossip, perform better than the random function. Indeed, the random function may remove a notification from a cache from a replica server before this server has been able to select it for gossiping. Especially when this happens at an early stage of the dissemination process of a notification, such a notification might replicate slower, or even not reach all replica servers at all. As the other three functions give new notifications a low probability of being

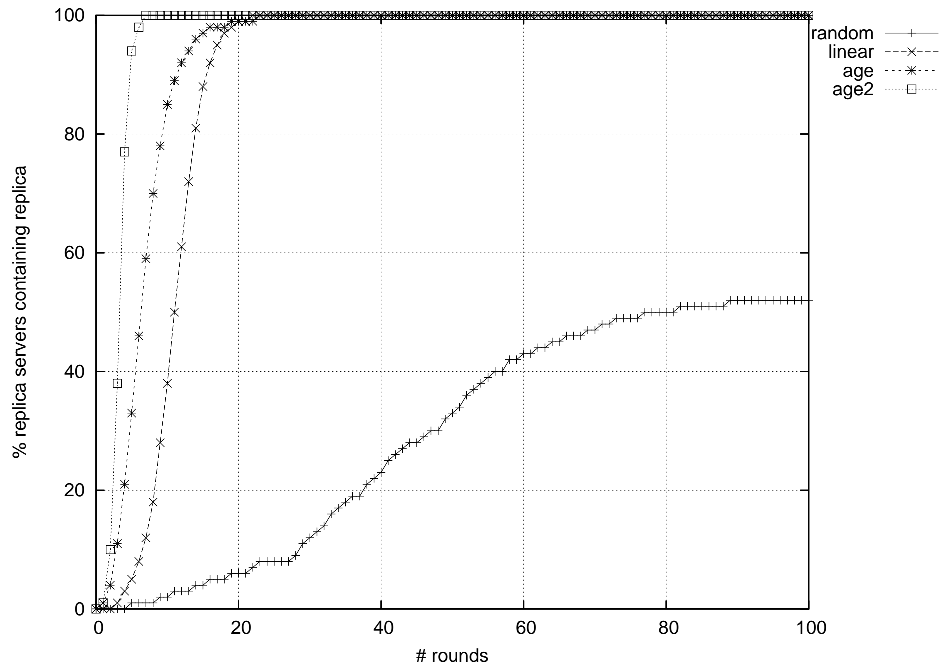


Figure 4.4: Dissemination progress of an insert notification using various select to send functions and RANDOM as select items to keep function. Insert notification inserted every five rounds.

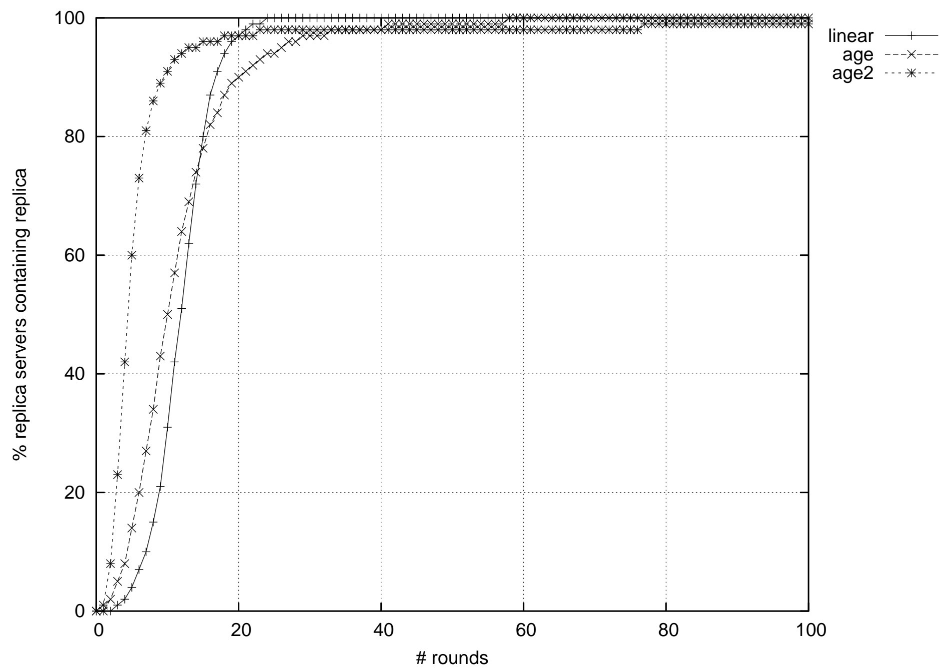


Figure 4.5: Dissemination progress of an insert notification using various select to send functions and RANDOM as select items to keep function. Insert notification inserted every two rounds.

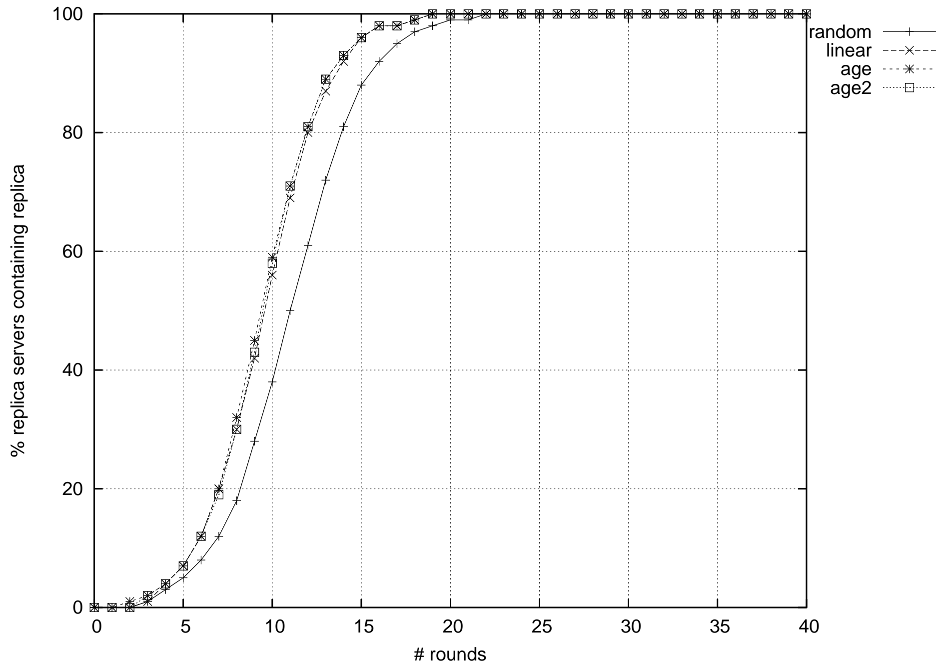


Figure 4.6: Dissemination progress of an insert notification inserted every five rounds using the best select to send function and various select items to keep functions.

removed from a cache, it is less likely that such a situation occurs with these functions.

Figure 4.6 shows the dissemination progress of an insert notification inserted every five rounds for the various select items to keep functions. We see that when we use *RANDOM* as select items to keep function, an insert notification reaches all replica servers in 22 rounds. The three other functions perform the same resulting in 19 rounds.

However, as shown in figure 4.7, when we insert an insert notification every two rounds, AGE2 performs slightly better than *AGE* and *LINEAR*. Therefore, we decide to use *AGE2* for our select items to keep function.

### 4.2.4 Threshold

In this section we investigate whether using a threshold for the select items to keep function improves the dissemination speed of the insert notifications. When we use a threshold of $x$ percent, this means we keep the freshest $x$ percent of the insert notification in cache anyhow when we apply the select items to keep function. So the select items to keep function works only on the other insert notifications in cache. This way, we may protect the most recent insert notifications until newer ones arrive, and increase the chance that we select them for sending before we remove them from a cache.

We run our experiments using a threshold of 10, 20, 30 and 40 percent. Figure 4.8 shows the dissemination speed of an insert notification inserted every five rounds using the various thresholds and figure 4.9 shows the same for notifications inserted every two rounds. We can see that the graphs are extremely similar. For all used thresholds, a notifications needs 20 rounds in order to reach all replica servers. Therefore, it does not make sense to use such a threshold. Apparently, the select items to keep function AGE2 gives the newest insert notifications such a high probability of being kept in cache that a threshold has no effect on the dissemination speed at all.

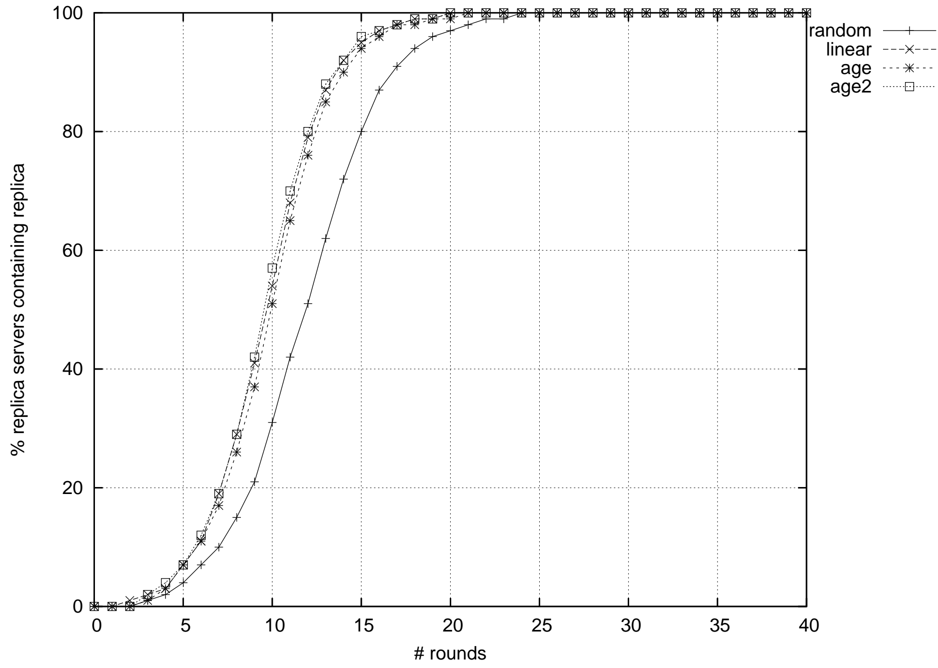


Figure 4.7: Dissemination progress of an insert notification inserted every two rounds using the best select to send function and various select items to keep functions.

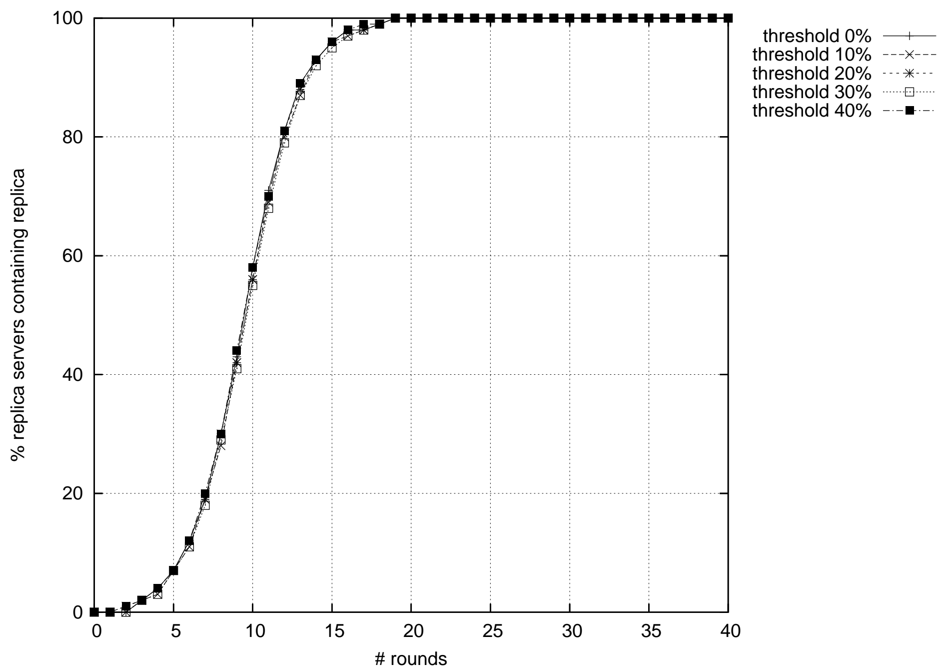


Figure 4.8: Dissemination progress of an insert notification inserted every five rounds using various thresholds.

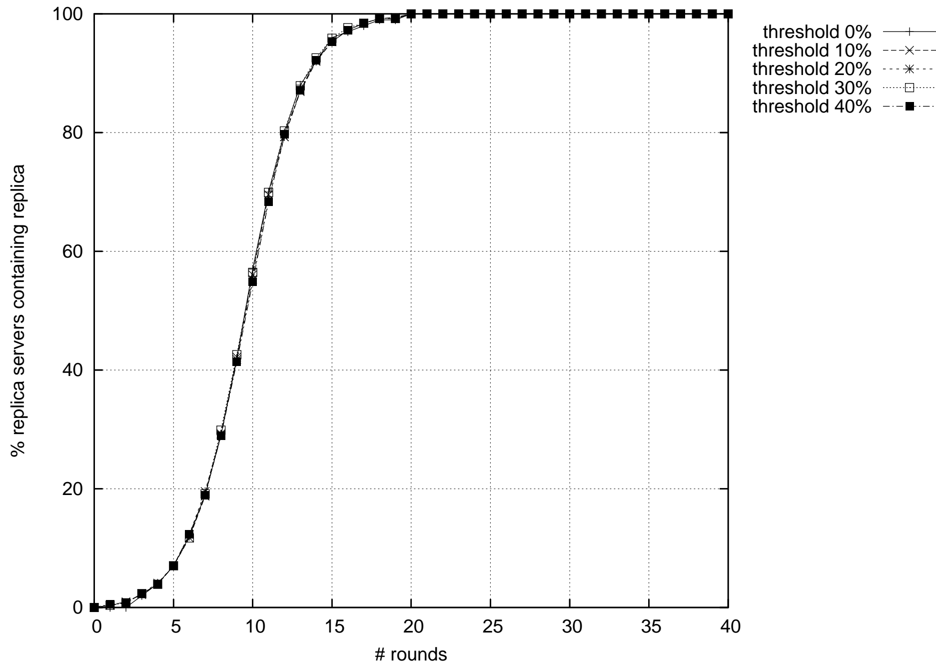


Figure 4.9: Dissemination progress of an insert notification inserted every two rounds using various thresholds.

### 4.2.5 Optimal cache size and gossip length

Finally, now that we have identified the best function for both selecting items to send and selecting items to keep, we would like to determine whether the initial cache size and gossip length are optimal or whether we can use a smaller value.

We run the same experiments as before, inserting 80 documents, one every two gossip rounds. We start with determining the optimal cache size for the server identifiers. Figure 4.10 shows the dissemination progress of an insert notification for server identifier cache sizes of 20, 10 and 5. We see that the cache size barely matters, and that the protocol is very robust to various cache sizes. In the rest of the experiments we set the server identifier cache size to 10.

Next, we determine the optimal cache size for insert notifications. Figure 4.11 shows the dissemination progress of an insert notification for notification cache sizes of 20, 10 and 5. We see that when we use a cache size of 10, a notification only needs 11 rounds to reach all replica servers, instead of 19 rounds. However, a cache size of 5 gives an even better result: a notification reaches all replica servers within 6 rounds. Therefore, we set the notification cache size to 5.

Note that this result is counter-intuitive as we would expect a faster dissemination progress of the notifications when we use a greater cache size. In a small cache, the probability of selecting new items is higher. Therefore, new items are selected for gossiping at least once before newer items arrive. This leads to a high dissemination speed of the notifications. However, a smaller cache size bounds the insert rate of the notifications. We presume that when we (temporarily) insert notifications at a higher rate, notifications will get removed from the cache before being selected for gossiping, leading to a smaller dissemination speed.

Now, we can determine the optimal gossip length for the server identifiers. Figure 4.12 shows the dissemination progress of an insert notification for server identifier gossip lengths 3, 2 and 1. We see that the results are the same for the three gossip lengths. This is due to the fact that in these experiments, the set of nodes participating in the system does not change. When nodes join and leave, a higher gossip length would be more appropriate. This is not the case in our experiments. Therefore, we decide to use the minimum value and thus set the server identifier

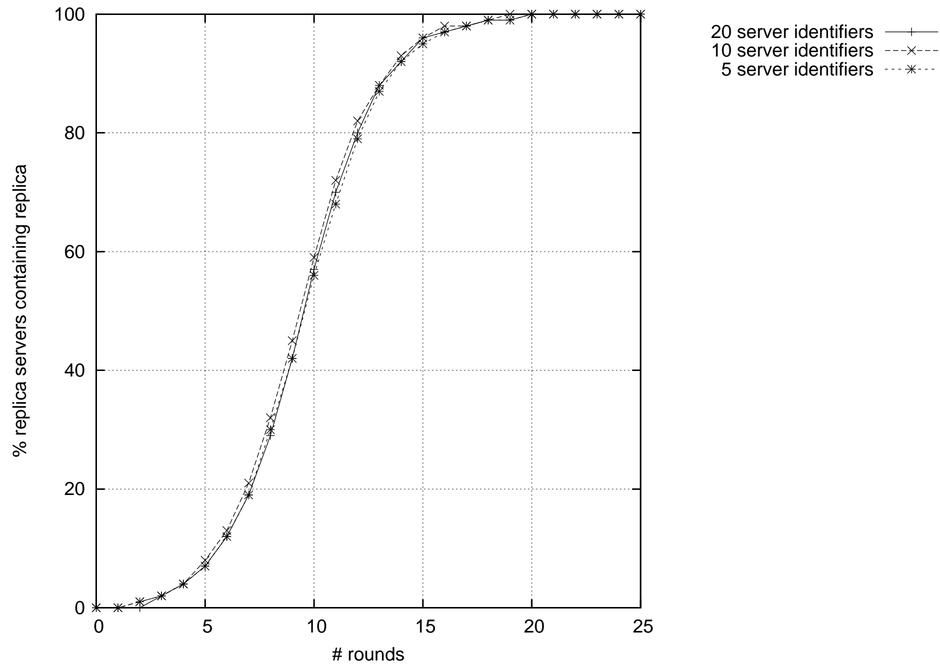


Figure 4.10: Dissemination progress of an insert notification for various server identifier cache sizes.

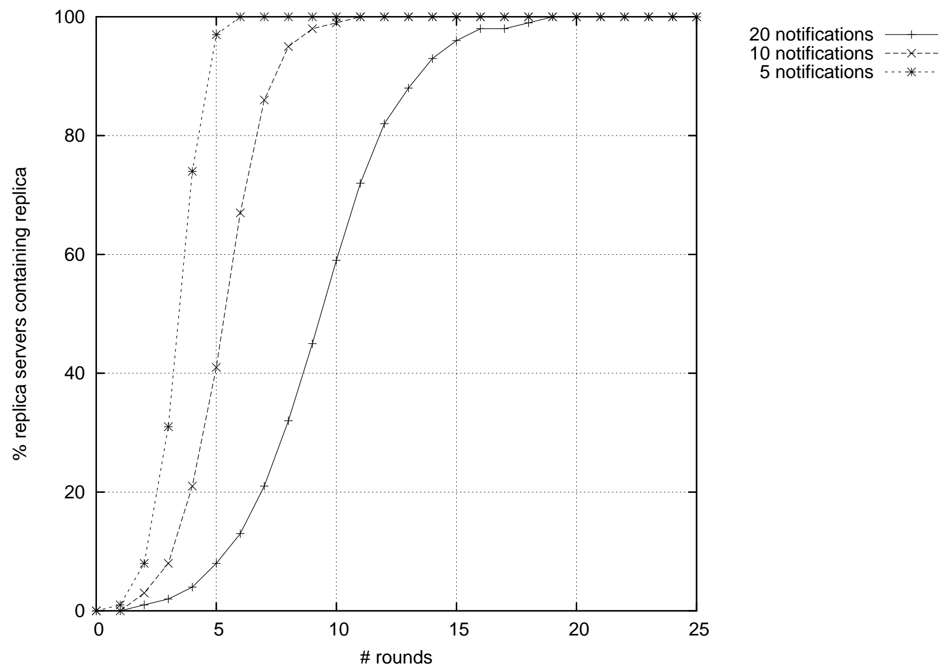


Figure 4.11: Dissemination progress of an insert notification for various insert notification cache sizes.

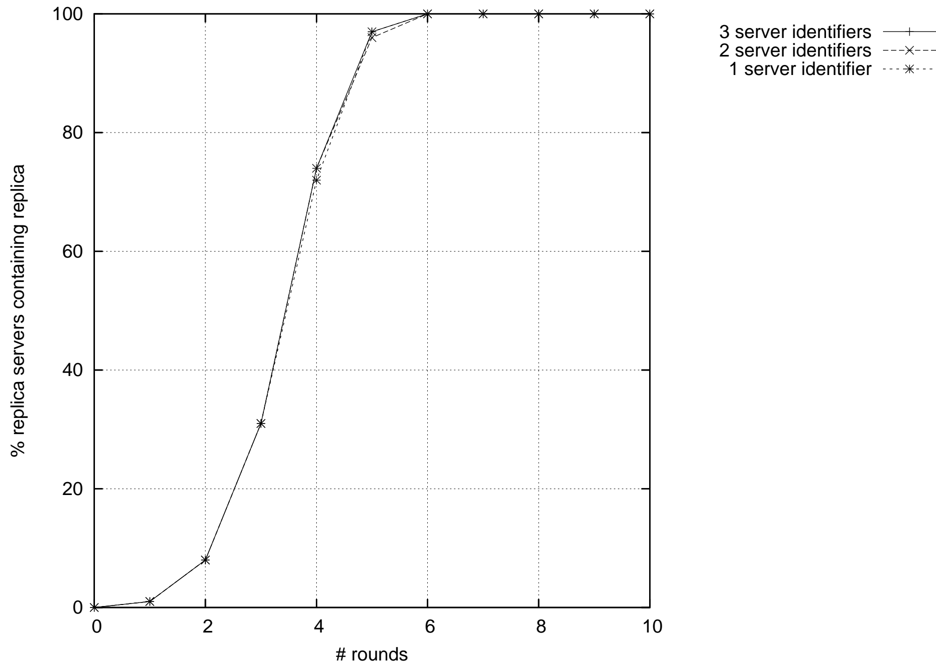


Figure 4.12: Dissemination progress of an insert notification for various server identifier gossip lengths.

gossip length to 1.

The last step is to determine the optimal gossip length for the insert notifications. Figure 4.13 shows the dissemination progress of an insert notification for notification gossip lengths 2, 3, 4 and 5. When we use gossip length 5, we exchange all notifications within the cache. We see that a gossip length of 2 (8 rounds) performs worse than the initial gossip length 3 (6 rounds). Increasing the gossip length to 4 decreases the number of rounds to 5. However, when we use a gossip length of 5, we obtain the same result as with gossip length 4. Therefore, we decide to use the gossip length 4 for the insert notifications.

## 4.3 Conclusion

We have presented a decentralized full replication policy for Web documents. The main issue of this policy is to make sure all replica servers have an up to date copy of all documents in the system. To accomplish this, the policy needs to spread insert notifications across all replica servers in an efficient and decentralized way. We structure the policy following epidemic protocols as these protocols have the property of rapid and efficient dissemination of information.

We use CYCLON to keep the network connected and handle membership management, as it creates communication graphs with a low clustering coefficient, which is good for the connectivity of the overlay and decreases the probability of redundant message delivery. In addition, we add to CYCLON the ability to spread insert notification, similar to the way Newscast spreads news. According to our experiments, we can conclude that the best performing select to send function is *LINEAR*, and the best select to keep function is *AGE2*.

Using this policy, we are able to spread insert notifications across 128 replica servers in about 5 rounds, using a cache size of 10 server identifiers and 5 insert notifications, and a gossip length of 1 server identifier and 4 insert notifications.

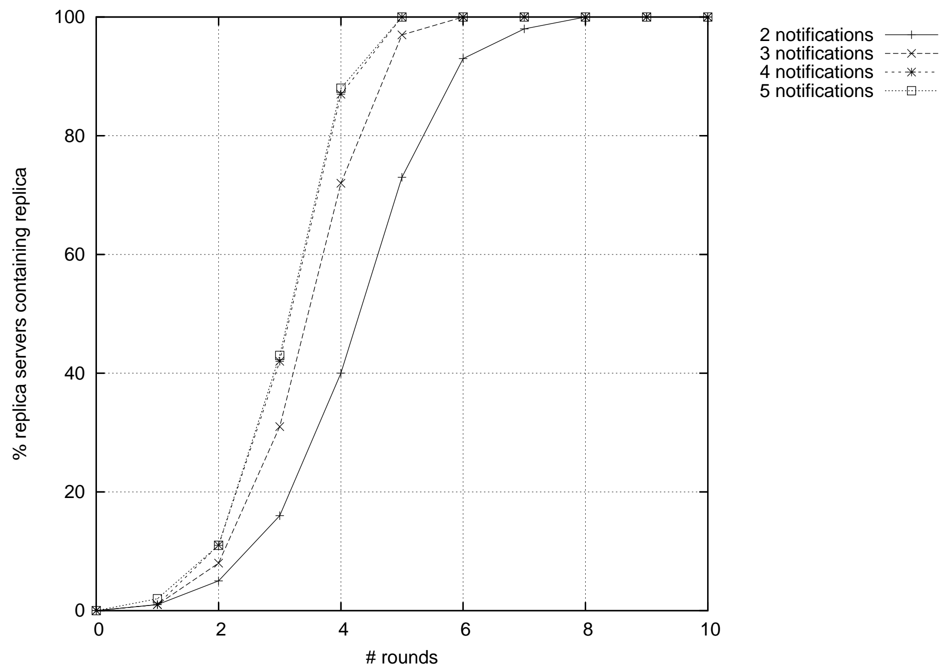


Figure 4.13: Dissemination progress of an insert notification for various insert notification gossip lengths.

# Chapter 5

# A Partial Replication Policy

In this chapter we present a decentralized replication policy for Web documents that allows for controlled partial replication. With partial replication, the replicas of a Web document are spread across $k$ replica servers in a network with $N$ nodes and $0 < k \leq N$.

A partial replication policy must address more issues than a full replication policy. First, instead of spreading the replicas of a document to all replica servers in the system, a partial replication policy has to make sure it places replicas at exactly $k$ replica servers. To accomplish this, we can add a counter to the insert notifications, which we decrease every time we create a replica at a replica server. However, when we replicate insert notifications, we have to take care of the counter, for example by splitting the counter between the sending and receiving server, otherwise we may place too many replicas of a document.

Another issue is how we can keep the replicas of a document consistent in the presence of document updates. We place replicas at random replica servers as discussed in Section 2.2, therefore replicas of an updated document may be created at different replica servers than the replicas of the previous version. We therefore have to take measures in order to inform such replica servers that they need to drop the stale replica.

Locating replicas is also an important issue for a partial replication policy, because in the case of partial replication, a client may request a document from a replica server that does not have a local copy of the document. In that case, a replica server must be able to locate a replica server that does have a local copy.

Finally, the replication policy must be able to deal with leaving and failing nodes, and network failures.

We present two versions of the partial replication policy. The first one is based on the full replication policy presented in the previous chapter. The second one is based on the structured peer-to-peer system Chord. The advantage of the former is that maintenance costs for the overlay network are low. However, locating replicas may be difficult as there is no correlation between replica servers and the content. As our experiments will show, this policy does not perform well enough in locating replicas. Therefore, we also present an improved version of the unstructured policy that uses a two-layered approach in order to improve locating replicas. With the structured version, locating replicas is easy because it creates a structured overlay network and places documents at specific replica servers. However, maintenance of the overlay network may become costly.

The main goal of this chapter is to optimize both partial replication policy versions and determine how they differ in performance and costs.

Section 5.1 presents the unstructured version of the partial replication policy based on the full replication policy discussed in the previous chapter. We evaluate this policy in Section 5.2. Section 5.3 describes the improved unstructured replication policy that is based on a two-layered approach. Section 5.4 evaluates this improved version. In Section 5.5, we present the structured version of the replication policy that is based on Chord. We evaluate the structured version in

Section 5.6. Finally, section 5.7 compares the two versions in both performance and maintenance costs.

## 5.1 Unstructured version

The unstructured version of the partial replication policy is based on the full replication policy that is structured following epidemic protocols as we described in the previous chapter. In this section we discuss the issues that arise when we use the full replication policy in order to achieve partial replication.

### 5.1.1 Replica placement

We need to place replicas of a document at exactly $k$ replica servers, instead of placing a copy of a document at all replica servers. Therefore, we add a *replication counter* to the insert notifications. We decrement this counter every time we create a new replica of the corresponding document on a replica server. In the full replication policy, we replicate the insert notifications across all replica servers.

However, when we replicate the insert notifications with replication counter across the replica servers, we have to take care of adjusting the replication counter. If we simply copy the counter when we replicate an insert notification, we will create too many replicas. An option is to split the counter equally between the sending replica server and the server that receives the insert notification. However, a potential problem is that this solution increases the number of insert notifications in the system every gossip round. Furthermore, at the end of a gossip we can only remove insert notifications with a counter equal to zero from the cache. If we remove an insert notification with a counter greater than zero, this results in placing less than $k$ replicas in the system.

Another solution is to migrate the insert notifications across the replica servers. The advantage is that per document version the system contains maximum one insert notification and thus one replication counter. This way, we can easily adjust the replication counter without an increase of insert notifications. A disadvantage of this solution is that migrating insert notifications generally takes more gossip rounds than replicating them. Fortunately, often we do not have to place replicas at all replica servers in the system. Therefore, we use migration in our experiments to spread the insert notifications.

### 5.1.2 Consistency enforcement

Another issue is keeping the replicas of a document consistent in the presence of document updates. As we place replicas at randomly selected replica servers, we may place replicas of an updated document at other replica servers than the replicas of the previous version. The replication policy has to inform those replica servers that they have to drop the stale document versions. However, because of the random placement, we do not know which replica servers possess a stale replica. Therefore, we have to inform all replica servers in the system that they have to drop the stale document version if present. For this, we can use the full replication policy in order to spread these *remove notifications* across all replica servers in the system.

### 5.1.3 Replica location

When a client requests a document from a replica server that does not have a replica stored locally, the replica server has to locate it. This is not an easy task as replicas are placed randomly in the system. A simple solution is for the requested replica server to initiate a recursive search through the server network, similarly to the Gnutella protocol. In such a case, we have to

determine how many hops it takes on average to find a replica that is not stored locally. Section 5.3 will investigate a possible improvement to this very simple policy.

### 5.1.4 Availability

Finally, servers may leave or fail, and parts of the network can be overloaded. In addition, new servers may join the system and failing servers may recover.

When a replica server wants to join the system it has to perform the CYCLON join operation (Section 3.2) in order to become part of the overlay network. In addition it could take over replicas from other, possibly heavily loaded, replica servers.

To leave the network a replica server can just stop gossiping. This way it will be forgotten in a limited amount of time. In addition it could send its replica list to a server from its cache and make that server responsible for creating these replicas on other replica servers in the system.

In case a replica server fails it will be forgotten in a limited amount of time just like with a leaving server. It is not possible to determine which replica it hosted. During the time it is unavailable its replicas are just not accessible.

When a replica server recovers there are two possible situations. One, the server has no state and should just act as a joining server. Two, it has does have a state and should ask one or more other replica servers for missed updates.

### 5.1.5 Maintenance

In order to keep the network connected the servers in the network have to exchange server identifiers from their caches. In the previous chapter we determined that exchanging one server identifier per gossip round is enough to keep a network of 128 nodes connected.

We need to spread remove notifications in order to remove stale replicas. As we place the replicas at random replica servers we need to disseminate the remove notifications to all replica servers to make sure all replica servers with a stale version are reached. We use the full replication policy for this purpose. So, every gossip round a replica server exchanges four remove notifications.

Finally, we disseminate insert notifications to spread the announce of new documents and updates. We use the same gossip length as for the remove notifications.

## 5.2 Performance evaluation unstructured version

We ran our experiments on a simulator implementing the unstructured partial replication policy. It simulates 128 replica servers and an origin server. We use the same cache sizes and gossip lengths as determined in the previous chapter. We use the same values for the remove notifications as for the insert notifications. Before we insert any documents into the system, the simulator runs the same warm-up phase as the full replication policy (Section 4.2), in order to create a connected overlay network. Next, we insert some warm-up documents and updated versions, in order to fill up the remove notifications part of the caches. The insert notifications part of the caches will be almost empty when we start our experiments as we remove an insert notification from a cache when its replication counter hits zero.

We insert a document into the system by adding a corresponding insert notification to the cache of the origin server. When the document is an update, we also add a remove notification to the cache in order to remove stale documents from the system. A replica server that receives an insert notification, fetches the corresponding document from the sender of the notification when it does not have it yet, and decrements the counter of the notification.

For our experiments we use a trace file of all document accesses and updates from the VU Web server. The trace runs from November 13th to November 20th 2005. We filtered out all

Table 5.1: Trace results.

| nr replicas | 4 | 8 |
|---|---|---|
| nr requests | 1,270,183 | 1,270,183 |
| nr retrieves | 454,806 | 708,892 |
| % retrieves | 36 | 56 |
| nr misses | 815,377 | 561,291 |
| % misses | 64 | 44 |
| nr stale | 3,209 | 4,910 |

requests for documents that are requested less than 50 times. We did this as replication makes sense only for relatively popular documents and in order to keep the simulator manageable. The filtered trace file consists of 3,937 documents, 1,270,183 requests and 4,815 updates. The number of updates is excluding the initial creations of the documents.

We use the trace file as follows. We execute two runs. The first run consists of analyzing the trace file and spreading the replicas of the documents with a last modification date smaller than the start time of the second run. We set the start time of the second run to the request date of the first request. In the second run we issue the requests and insert the remaining replicas. During each gossip round we first insert all replicas of documents of which the last modification date falls within the current round. Next, we issue the requests with a request date that fall within the current gossip round. At the end of a gossip round each server initiates a gossip. We set the gossip round to 10 seconds.

### 5.2.1 Requests

We first want to determine how many hops it takes to serve a client request for a document. We run experiments where we place 4 and 8 replicas of each document in the system. When a server reveives a client request it first searches in its local documents. In case it does not have the requested document locally, the server asks the servers in its cache one by one without recursing the search further.

Table 5.1 gives an overview of the results. We see that when we place 4 replicas only 36 percent of the request can be served. When we place 8 replicas the number of successfull requests increases to almost 56 percent. However, these results are still far from ideal.

A possible solution is to increase the cache size, or extend the search algorithm by letting the servers in the cache of the requested replica server forward the request to the servers in their caches. However, this may lead to flooding the network with search messages. Therefore, we propose a different solution. We create a second layer on top of the *CYCLON* layer that facilitates searching. We discuss this solution in the next section.

## 5.3 Improved unstructured version

As we have shown in the previous section, the unstructured partial replication policy does not perform well enough. It takes way too many hops for a replica server to locate a replica that is not stored locally. Therefore, we propose a two-layered approach.

The bottom layer runs the full replication algorithm. It is responsible for membership management and keeping the network connected. In addition, it spreads remove notifications to all replica servers to remove stale documents. On top of this layer we create a second layer that facilitates searching for replicas in the network. We use T-Man to construct the overlay network into a torus (as discussed in Section 3.2), which is a suitable topology for searching

purposes. In our experiments, we assign all servers and documents a random 2-dimensional identifier consisting of two 31-bit attributes. Another way to obtain such an identifier is hashing a server's address and a document's URL. The distance function is defined as the sum of the distances in both directions with applying the periodic boundary condition. So, the distance in one dimension is defined by $d(a,b) = min(N - |a-b|, |a-b|)$, where $a$ and $b$ are two points from an interval $[0, N]$. Ranking is defined through this distance function. Each node can order the server identifiers in its cache according to increasing distance from its own identifier.

Inserting replicas of a document requires routing an insert notification to a node with an identifier close to the identifier of the document. We achieve this by forwarding the notification to the server in the current cache (from both layers) with an identifier closest to the identifier of the document, as long as the current cache contains a server identifier closer than the identifier of the owner of the cache. When the notification has reached the closest server, this server fetches the corresponding document from the origin server decrements the notification counter, and adds the notification to its top layer cache.

During a gossip round of the top-layer, server identifiers are exchanged in order to maintain the overlay topology. We apply the ranking function on all server identifiers from both layers and randomly select gossip length server identifiers from the first top-layer cache size items. We select a gossip partner in the same way as T-Man: applying the ranking function on the server identifiers from the top layer and selecting a random server from the first half. In addition, the servers exchange insert notifications. This way we spread the insert notifications across replica servers with an identifier close the identifier of the server that initially received the notification, and thus close to identifier of the document. We use $LINEAR$ to select the insert notifications. Selected insert notifications are removed from the cache, in order to migrate them just like with the previous version. At the end of a top-layer gossip, a server applies the ranking function on the server identifiers from both layers and keeps the top-layer cache size first items in its top-layer cache. Furthermore, it keeps the received insert notifications and removes all notifications with a counter equal to zero.

When a client requests a document from a replica server that does not host the requested document, the server needs to locate it. It can find the document by forwarding the request to the server in its cache (from both layers) with an identifier closest to the identifier of the document until it reaches a server that has a local copy.

## 5.4 Performance evaluation improved unstructured version

We ran our experiments on a simulator implementing the two-layered unstructured partial replication policy. It simulates 128 replica servers and an origin server. Each experiment starts with a warm-up phase. In this warm-up phase, the bottom layer creates a connected overlay and fills up the remove notifications part of the cache in the same way as the previous version of the unstructured partial replication policy. A gossip round starts with each server initiating a bottom-layer gossip. After that, each server initiates a top-layer gossip for constructing the torus topology.

The cache of the top-layer consists of 20 server identifiers and 10 insert notifications. The gossip length is 3 for the server identifiers and 4 for the insert notifications. We use the same trace file as we used for the previous unstructured version.

### 5.4.1 Inserts

We first want to determine how many hops it takes to insert the replicas of a document into the system. Figure 5.1 gives an overview of the number of hops it takes to insert a replica. We see that the median number of hops is 3. This is due to the fact that it takes about 3 hops in order to find the node with an identifier closest to the identifier of the document. Note this is

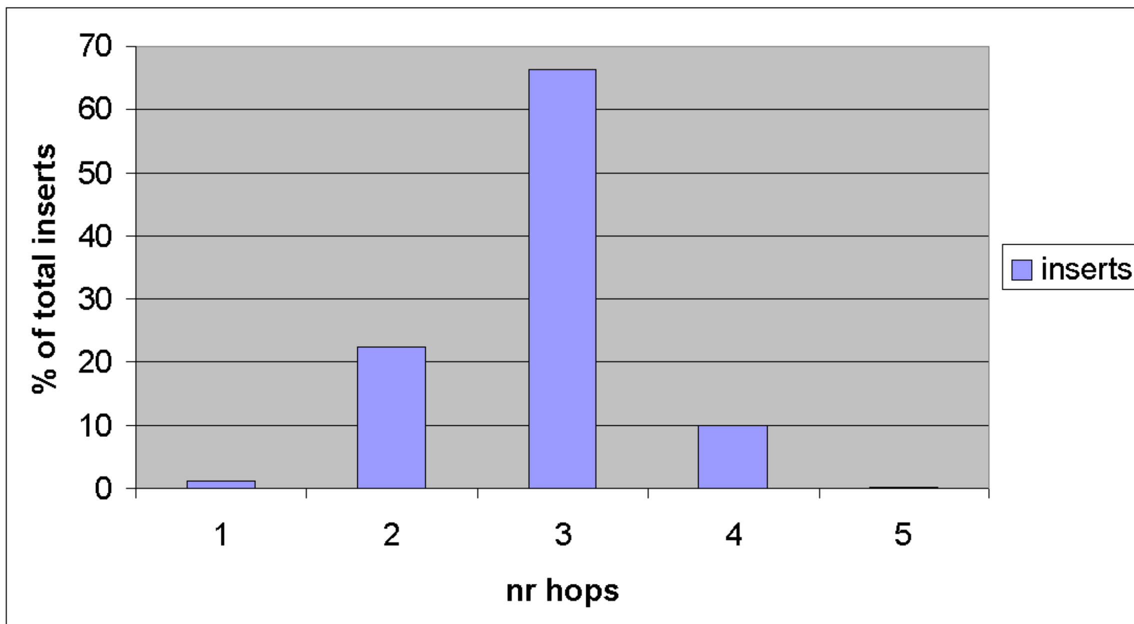


Figure 5.1: Number of hops per insert.

Table 5.2: Results of all requests.

| Nr replicas | Average nr hops | Median nr hops |
|---:|---:|---:|
| 4 | 2.72 | 3 |
| 8 | 2.60 | 3 |
| 16 | 2.44 | 2 |

the number of hops it takes to insert one replica. Inserting $k$ replicas of a document requires an additional $k-1$ hops. Furthermore, that takes at least $k-1$ gossip rounds because we insert the additional replicas by migrating the insert notification using gossiping to the remaining replica servers.

### 5.4.2 Requests

We also want to determine how many hops it takes to serve a client request for a document. We run experiments where we place 4, 8 and 16 replicas.

Figure 5.2 shows the distribution of the number of hops it takes to serve a client request for a document. Table 5.2 gives an overview of the average and median number of hops. We see that it takes about three hops to serve a client's request when we place 4 or 8 replicas. Placing 16 replicas results in a median hop count of two.

Figure 5.3 gives an overview of the number of hops it takes to serve a client request in percentage of requests that cannot be served locally. Table 5.3 shows the average and median number of hops. We see that requests for documents that are not stored on the requested replica server take about three hops. These results are considerably better than those obtained with the first unstructured partial replication design.

We note that searching for a partially replicated document is slightly faster than inserting it. When we place 16 replicas about 50 percent of the requests are served in less than three hops, whereas inserting a replica requires only less than three hops in about 25 percent of the inserts. So placing replicas does not only improve availability, which is our main reason for replication,

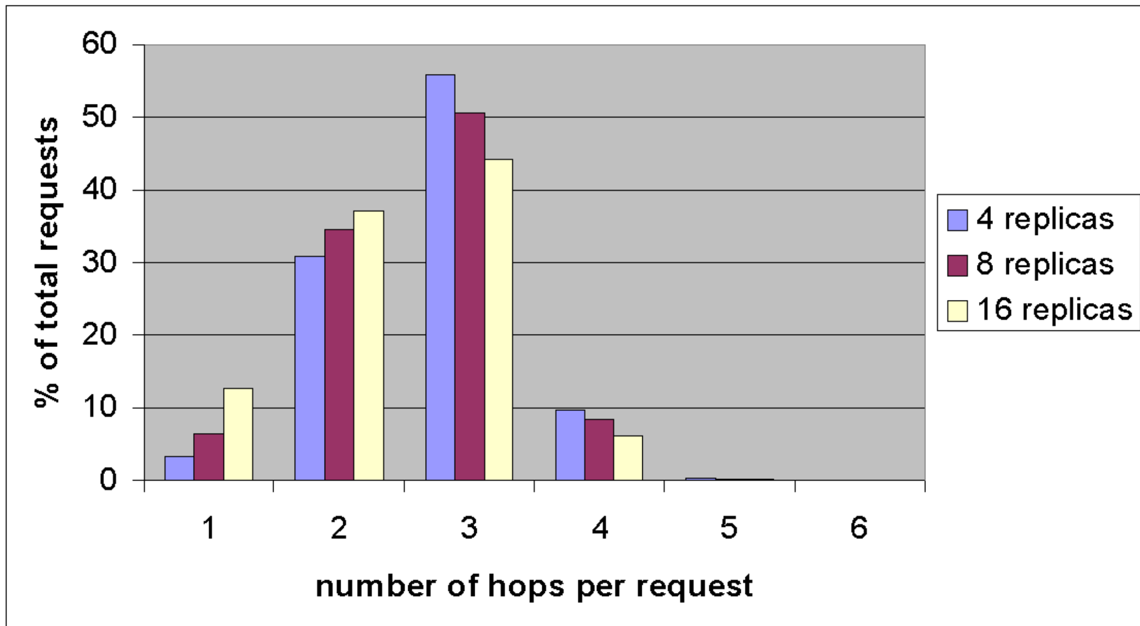


Figure 5.2: Distribution of the number of hops per request.

Table 5.3: Results of requests not served locally.

| Nr replicas | Average nr hops | Median nr hops |
|---|---|---|
| 4 | 2.78 | 3 |
| 8 | 2.71 | 3 |
| 16 | 2.65 | 3 |

but also performance, which is a nice side-effect.

### 5.4.3 Costs

During each gossip round, the replica servers need to exchange data in order to keep the network connected (bottom-layer) and maintain the torus topology (top layer). In a gossip round, each replica server initiates two gossips, one for each layer. A server exchanges one server identifiers in a bottom-layer gossip and. three in a top-layer gossip.

Maintenance of the overlay network thus requires sending $4 * N$ messages containing four server identifiers per gossip round for a network of size N. A server identifier consists of a server address and a timestamp. In addition, to this messages we can add notifications in order to spread information about new documents and updates.

## 5.5 Structured version

The structured version of the partial replication policy is based on Chord [23]. In this section we discuss the issues that arise when we structure our partial replication policy following Chord.

### 5.5.1 Replica placement

We need to place replicas of a document at exactly $k$ replica servers. We use Chord to determine the successor node (node whose identifier is equal to or follows the identifier of the document in

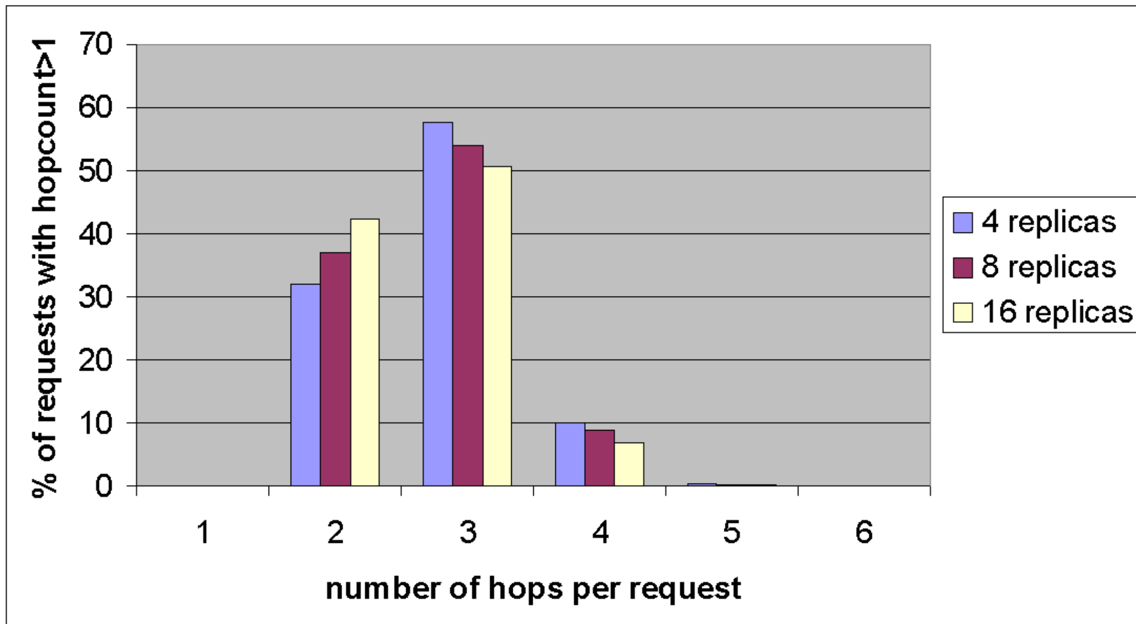


Figure 5.3: Distribution of the number of hops per request not served locally.

the identifier space) of the document. Subsequently, we place replicas at the successor and the $k-1$ nodes that logically follow the successor node on the Chord identifier ring. To accomplish this, the origin server can send the successor node an insert notification that contains the name of the document, its last modification date, and a *replication counter* with value $k$.

We can spread the insert notification to the next $k-1$ nodes in two different ways. First, when a replica server receives an insert notification, it can fetch the corresponding document from the server that sent the notification, decrease the counter and forward the notification to its successor node. When the counter hits zero, a server can drop the insert notification.

Another option is to use a node's successor list in order to forward the insert notification. This way, the successor node can inform the servers in its successor list about the notification. When we make sure the successor list is greater than or equal to the number of replicas we need to place, it can inform all replica servers that need to place a replica of the document. An advantage of this approach is that we can make the successor node responsible for making sure its $k-1$ successor nodes have an up to date version of the document. Furthermore, whenever it notices a replica server in its successor list has failed, it can remove this server from the list, add a new successor node and send a notification to this node. However, when the number of replicas is large compared to the number of nodes in the network, the successor list becomes large and its maintenance cost may increase. Note that this option imposes a system-wide higher bound on the document's replication degree. In the first option, each document could potentially be given an arbitrary replication degree with no restriction.

In our experiments, we use the first option as we do not run experiments with failing servers and do not want to restrict the replication degree.

### 5.5.2 Consistency enforcement

Another issue we need to take care of is keeping the replicas of a document consistent in the presence of updates. We place a document on its successor node and the $k-1$ logically following nodes. An update of a document gets the same identifier as the previous version. Therefore, they have the same successor node. So, in case of an update the successor node can simply drop the old version and replace it by the new one. We discuss maintaining consistency in the

presence of joining and leaving servers in Section 5.5.4.

### 5.5.3 Replica location

When a client requests a document from a replica server that does not have a replica stored locally, the replica server has to locate a replica. The replica server can find the successor node of the document using Chord.

However, Chord always finds the first successor of the document and thus we always issue the same replica server for a given document. While this is correct with respect to replication for fault tolerance, it will usually lead to an unbalanced load. Another option is to first find the predecessor node (first node that precedes the successor node in the identifier space) of the document, and ask this node for a random successor node from its successor list. However, this must be a node that has the requested replica stored locally. This means, the predecessor node has to know how many replicas the requested document has, or may have to contact multiple nodes from its successor list, until it has found one with the requested document.

In our experiments, we issue the request to the first successor of the document as we are not interested in load balancing yet. We first want to determine how many hops it takes to serve a client request.

### 5.5.4 Availability

Finally, servers may leave or fail, and parts of the network can be overloaded. In addition, new servers may join the system and failing servers may recover.

When a server $n$ joins the system it has to find a server $m$ that is already part of the network. It can for example ask a well known bootstrap server for such a server. Next, $n$ should ask $m$ to find its successor node and store it in its routing table. Furthermore, server $n$ becomes the new successor for some of the documents of its successor node and should fetch these. In addition, a server that hosts the $k$th replica of such a document should drop it. We can handle this by storing a counter with each replica stating how many replicas there are left among the next successor nodes. Thus, the first replica stores a counter value of $k$ and the counter of the last replica is equal to zero.

When a server leaves the system, it can inform its successor about this. In addition it can inform its successor about its new predecessor node. The successor node becomes responsible for the replicas of the leaving server. If the successor already hosts a replica of the leaving server this information should be forwarded to the successor's successor until a server is found that does not have the replica yet. That server should now also host a copy.

When a node detects the failure of a node $n$ it can inform the successor node of $n$. The replica list of the failing node can be reconstructed by examining the replica list of its predecessor node. The failing server hosted a copy of all documents in this list with a counter value greater than zero.

A server that recovers from a failure can act as a joining server.

### 5.5.5 Maintenance

The structured overlay requires maintenance as servers may join and leave the system. The maintenance protocol consists of four steps. In the first step, a server checks if there is a newly joined server that is its successor node instead of the server that is currently stored as successor in the routing table.

The second step consists of initializing and checking the finger table. During a maintenance round a server initializes or checks one of its finger table entries by looking up the entry's successor node. The third step is checking whether the predecessor has failed.

Table 5.4: Results of all requests.

| Nr replicas | Average nr hops | Median nr hops |
|---:|---:|---:|
| 4 | 4.21 | 4 |
| 8 | 4.07 | 4 |
| 16 | 3.75 | 4 |
| 32 | 3.30 | 4 |
| 64 | 2.42 | 2 |
| 128 | 1 | 1 |

Finally, a server maintains its successor list. It copies the successor list of its successor node, removes the last entry, and prepends the successor node to it.

## 5.6 Performance evaluation structured version

We ran our experiments on a simulator implementing the structured partial replication policy based on Chord. It simulates 128 replica servers and an origin server. We assign all servers a random 63-bit identifier from the identifier space. (This is long enough for our experiments. Should we use more nodes and/or documents, a 128-bit identifier could be more appropriate.) We start all experiments with a warm-up phase in order to create a connected overlay. In this warm-up phase, the servers join one by one. The join operation of node $N$ consists of asking a bootstrap server for a random server $P$ that is already part of the network. In addition, node $N$ requests node $P$ to find the successor node of $N$, and stores this in its routing table. During this warm-up phase, the nodes periodically run a maintenance protocol in order to complete the routing tables. We discuss this maintenance protocol in Section 5.5.5.

We use the same trace file as we used for the unstructured version (Section 5.2). We assign all documents that are requested also a random 63-bit identifier.

### 5.6.1 Requests

We first want to determine how many hops it takes to serve a client request for a document. We run experiments where we place 4, 8, 16, 32, 64 and 128 replicas of all documents.

In these experiments the successor list contains only the first successor of a node. When a successor node receives a notification, it fetches the corresponding document from the server that sent the notification, it decrements the counter, and it forwards the notification to its successor node in case the counter is greater than zero.

Figure 5.4 shows the distribution of the number of hops it takes to serve a client request for a document. Table 5.4 gives an overview of the average and median number of hops. We see that when we place up to 32 replicas it takes about four hops to serve a client's request. When half of the replica servers have a replica of the documents it takes only two rounds. Obviously, when the documents are fully replicated all documents can be found locally resulting in a hop count of one. We see that the number of hops it takes to retrieve a document remains constant until we place more than 32 replicas.

Figure 5.5 gives an overview of the number of hops it takes to serve a client request in percentage of requests that cannot be served locally. Table 5.5 shows the average and median number of hops. We see that requests for documents that are not stored on the requested replica server take about four hops when we do not apply full replication. We conclude that in a network of 128 replica servers placing more than four replicas does not improve the number of hops it takes to serve a client request.

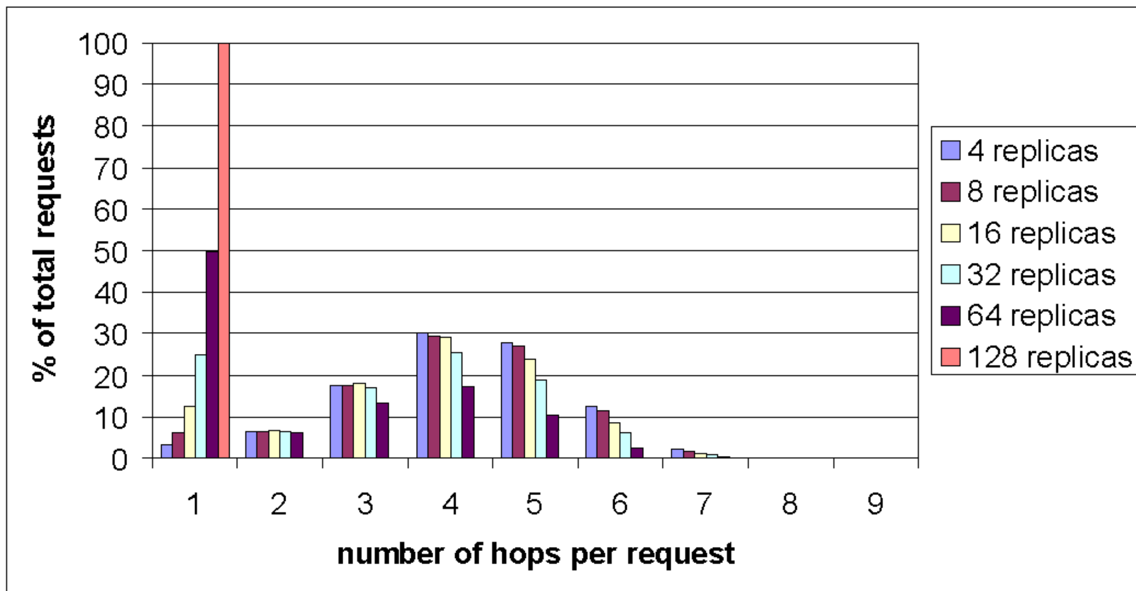


Figure 5.4: Distribution of the number of hops per request.

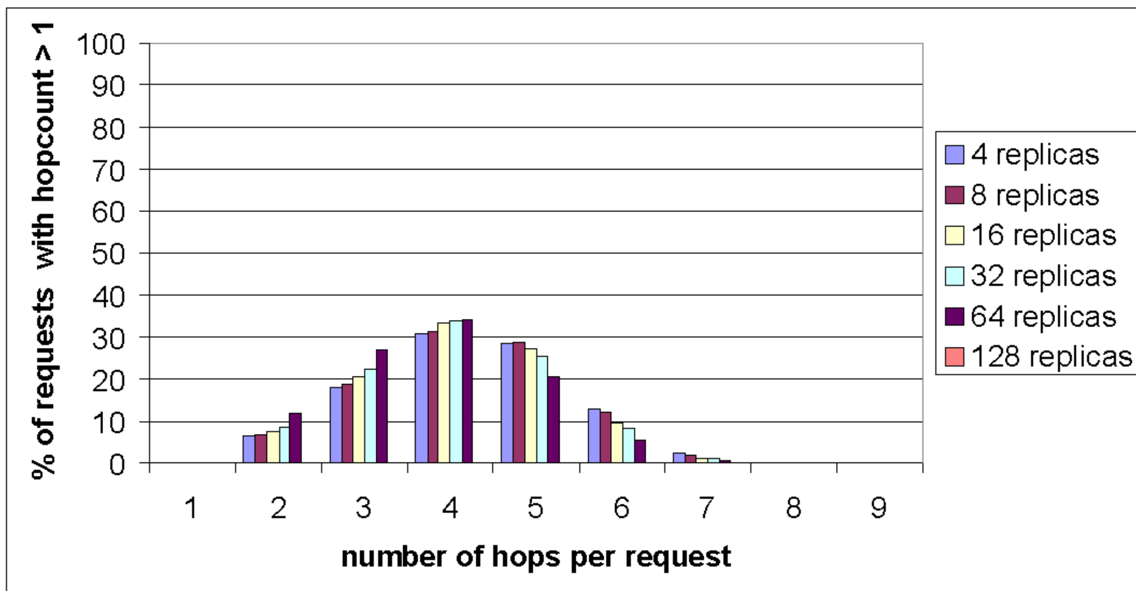


Figure 5.5: Distribution of the number of hops per request not served locally.

Table 5.5: Results of requests not served locally.

| Nr replicas | Average nr hops | Median nr hops |
|---:|---:|---:|
| 4 | 4.31 | 4 |
| 8 | 4.27 | 4 |
| 16 | 4.14 | 4 |
| 32 | 4.06 | 4 |
| 64 | 3.82 | 4 |
| 128 | - | - |

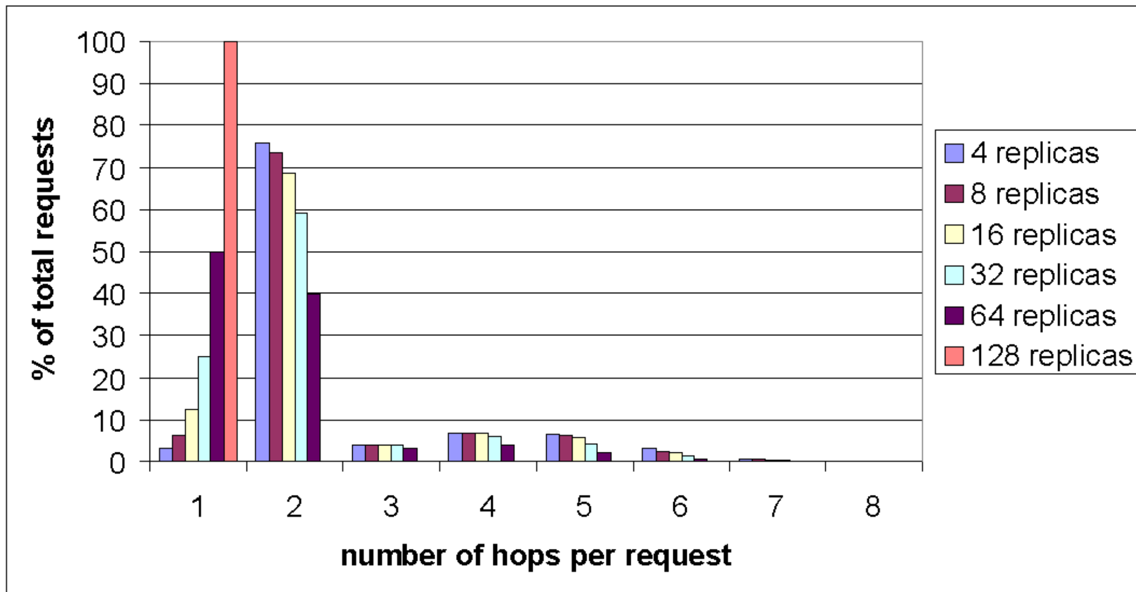


Figure 5.6: Distribution of the number of hops per request using references.

Table 5.6: Results of all requests using references.

| Nr replicas | Average nr hops | Median nr hops |
|---|---|---|
| 4 | 2.49 | 2 |
| 8 | 2.43 | 2 |
| 16 | 2.32 | 2 |
| 32 | 2.11 | 2 |
| 64 | 1.71 | 2 |
| 128 | 1 | 1 |

### 5.6.2 References

Now that we have determined how many hops it takes to serve a client's request, we would like to see how the use of references can improve this number of hops. Every time a replica server has to search for a document, it stores a reference to the server where it found the replica. A replica server examines its references first before using the search method. We do not set an upper bound on the number of references. A replica server can just store one reference per document. There are niftier caching methods available but these are out of the scope of this thesis and can be used in future research.

Figure 5.6 shows the number of hops it takes to serve a client request for a document in percentage of the total requests. Table 5.6 gives an overview of the average and median number of hops. We see that it takes about two hops to serve a client request.

Figure 5.7 gives an overview of the number of hops it takes to serve a client request in percentage of requests that cannot be served locally. Table 5.7 shows the average and median number of hops. We see that request for documents that are not stored on the requested replica server take about two rounds when we do not apply full replication. We conclude that using references can decrease the required number of hops to serve a request by two.

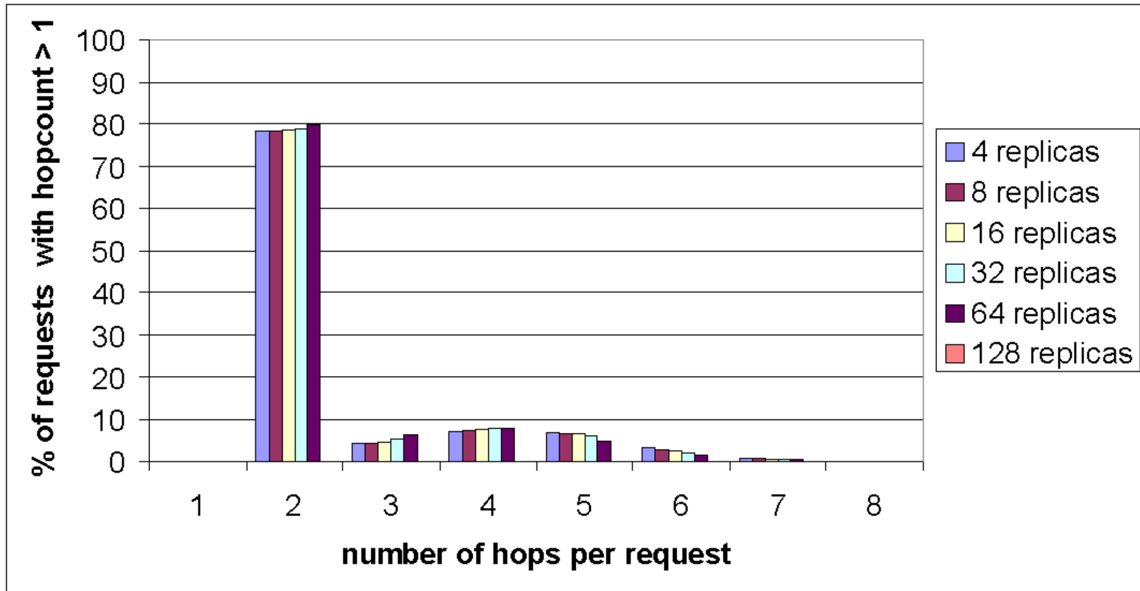


Figure 5.7: Distribution of the number of hops per request not served locally using references.

Table 5.7: Results of requests not served locally using references.

| Nr replicas | Average nr hops | Median nr hops |
|---:|---:|---:|
| 4 | 2.54 | 2 |
| 8 | 2.53 | 2 |
| 16 | 2.51 | 2 |
| 32 | 2.47 | 2 |
| 64 | 2.42 | 2 |
| 128 | - | - |

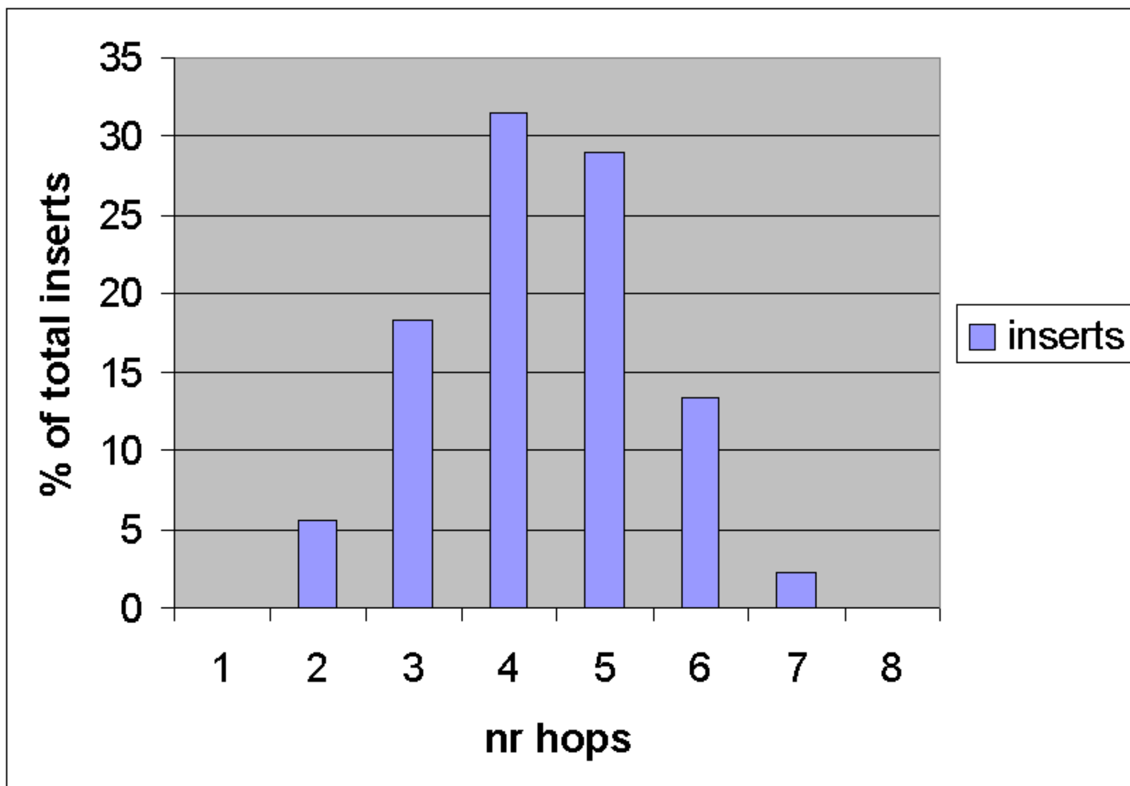


Figure 5.8: Distribution of the number of hops per insert.

### 5.6.3 Inserts

Another interesting issue is how many hops it takes to insert the replicas of a document into the system. Figure 5.8 gives an overview of the number of hops it takes to insert a replica. We see that the median number of hops is 4. This is the same as the number of hops it takes to locate a document. So, we can conclude that it takes 4 hops in order to find the successor of a document. Note this is the number of hops it takes to insert one replica. Inserting $k$ replicas of a document takes an additional $k$-1 hops.

### 5.6.4 Costs

During each maintenance round each server $n$ has to check if a new server has joined that is its new successor, by asking its successor node $m$ about the predecessor node of $m$. This requires sending two messages. If node $n$ does have a new successor node, $n$ should inform its new successor node that it is its new predecessor. This requires an additional message.

Furthermore, a server needs to fetch the successor list of its successor to update its own successor list. This leads to two additional messages.

Next, it should look up the successor node of one of its finger table entries. We determined that looking up a successor node takes about four hops and thus five messages.

Finally, a server needs to check if its predecessor node is still available. This requires two additional messages.

## 5.7 Comparison

Now we have evaluated the performance and costs of both the improved unstructured and the structured version of the partial replication policy, we compare the results in this section.

The improved unstructured version requires about three hops in order to insert the first replica, whereas the structured version takes about four hops. The structured version inserts the remaining $k-1$ replicas by forwarding the insert notification from successor to successor, leading to an additional $k-1$ hops. The unstructured version on the other hand, spreads the notification to the remaining $k-1$ replica servers during top layer gossiping. It may take some additional gossip rounds before all remaining replica servers are reached but it does not take extra messages.

Retrieving a document requires about three hops for the unstructured version, which is the same as for inserting. Although, searching for a document is slightly faster when we place 16 replicas. The structured version needs four hops, which is also the same as for inserting. However, if we cache document searches using references, documents can be found in only two hops.

It is hard to compare the maintenance costs of both versions. The unstructured policy has a clear maintenance protocol. Every gossip round, all replica server exchange data in order to maintain the network structure. Furthermore, the replica servers also exchange information about new documents, updates and stale versions at the same time. Thus, the number of messages exchanged per gossip round is stable, despite the rate in which servers join and leave the overlay network. Although, the presence of high churn may demand for more server identifiers to be exchanged per gossip round in order to adapt to these changes more quickly. But sending for example 10 instead of 5 server identifiers is not a big burden and at least the number of messages remain constant. The advantage of such a constant protocol is that we know the costs in advance and are able to prepare for that.

The maintenance protocol of the structured version is not that constant. First of all, in the structured version replica servers need to check their successor node, as invalid successor node references may lead to unsuccessfull searches. Furthermore, servers need to check their finger table entries, as invalid entries in this table may lead to slower searches. These checks require searches for successor nodes, are not constant and depend on the rate at which servers join and leave the system. Finally, it is not obvious how often servers need to do these checks. The maintenance round of the structured version does not necessarily have to be the same as that of the unstructured version.

So, performance results are almost equal and maintenance cost are difficult to compare. However, the main conclusion is that we created an unstructured partial replication policy that performs almost identical to a structured one. Furthermore, the unstructured version has the ability to quickly spread information to (a subset of) all servers in the network, and has constant maintenance costs. This are things the structured version lacks and makes the unstructured version an interesting alternative.

# Chapter 6

# Conclusion

The goal of this thesis was to design a decentralized replication policy for Web documents that assumes the origin server to be unavailable most of the time and only needs the origin server's presence to inform the replica servers about a new or updated document.

To accomplish this, we organized the replica servers in a decentralized fashion such that at least one of them has a copy of each document. In such a system, we must make sure replica servers know where they can find documents that they do not have stored locally. In order to support up to $n-1$ unavailable replica servers we made the replication policy tunable such that copies are available at $n$ different replica servers. Finally, to implement best-effort weak consistency we ensured that updates are spread in a reasonable limited time interval after an update takes place at the original document.

These properties come close to those offered by peer-to-peer overlays: sharing computer resources, decentralization, self-organization, resilience to network and server failures. Therefore, we decided to structure the replication policy along a peer-to-peer architecture.

We first introduced a replication policy that achieves full replication, which means that every replica server hosts a copy of all documents of the origin server. This simple version of a replication policy forms the base of a replication policy that achieves a more general form of replication, namely partial replication. The main challenge of the full replication policy is to spread notifications from the origin server about new or updated documents across all replica servers in an efficient and decentralized way. As these requirements are very similar to the properties of epidemic protocols, we structured the full replication policy following these protocols.

The full replication policy uses CYCLON to keep the network connected and handle membership management. We added the ability to spread notifications, similar to the way Newscast spreads news. Our experiments showed that *LINEAR* is the best performing select items to send function, and *AGE2* the best performing select items to keep function. *LINEAR* gives notifications a probability of being selected that decreases linearly with its age. *AGE2* gives newer notifications a significantly higher probabiliy of being selected than older ones.

With this policy we are able to spread notifications across 128 replica servers in about 5 gossip rounds, using a cache size of 10 server identifiers and 5 insert notifications, and a gossip length of 1 server identifier and 4 insert notifications.

Next, we presented a decentralized replication policy that allows for controlled partial replication. With controlled partial replication documents are replicated to exactly $k$ replica servers in a network with $N$ nodes and $0 < k \leq N$.

A partial replication policy must address more issues than a full replication policy. First, instead of spreading the replicas of a document to all replica servers in the system, a partial replication policy has to make sure it places replicas at exactly $k$ replica servers. Furthermore, it needs to keep the replicas of a document consistent in the presence of updates, as we may place replicas of an updated document at other replica servers than the replicas of the previous

document version. Finally, the policy must be able to locate replicas, as clients may request documents from replica servers that do not have a local copy of the requested document.

We created two versions of the partial replication policy. The first one is based on epidemic protocols and is an extended version of the full replication policy. We added a replication counter to the insert notifications to make sure we place replicas at exactly $k$ replica servers. We also introduced remove notifications that we spread to all replica servers using the full replication policy in order to remove stale documents. This was necessary, as an updated document may be placed at a different replica server than the previous version, and we need to inform all replica servers that host a stale copy that they need to drop them.

However, in this policy locating replicas becomes a problem. Initiating a recursive search through the server network, similarly to the Gnutella protocol, would require traversing many hops and may lead to flooding the server network with search messages. Therefore, we presented an improved version of the unstructured replication policy.

The improved version uses a two-layered approach. The bottom layer runs the full replication algorithm. It is responsible for membership management, keeping the network together and spreading remove notifications. The top layer uses T-Man to construct the overlay network into a torus such that it is suitable for searching purposes. In addition it spreads insert notifications to servers that are close with respect to their 2-dimensional identifier. With this policy, we were able to insert and retrieve documents while traversing only three server hops. These results are considerably better than those obtained with the first unstructured partial replication design. The maintenance protocol of the unstructured version has constant costs. During each gossip round a constant number of messages is exchanged between the replica servers. Furthermore, the replica servers also exchange information about new documents, updates and stale versions at the same time. The advantage of such a constant protocol is that we know the costs in advance and are able to prepare for that.

The second version of the partial replication policy is based on the structured peer-to-peer system Chord. It places replicas of a document on its first $k$ successor nodes. We determined that finding a document's successor node takes traversing about four hops. Therefore, inserting and retrieving documents required also about four hops. Although caching document searches using references can decrease the required number of hops to find a document by two hops. The costs of the maintenance protocol of the structured policy are not constant but depend on the rate at which servers join and leave the system.

The main conclusion of this thesis is that we were able to create an unstructured partial replication policy that performs almost identical to a structured one. We showed that an unstructured replication policy is not only good in terms of quickly spreading information, but can also be used for locating data. As the unstructured version has features that the structured one lacks, namely the ability to spread information quickly to (a subset of) all servers in the network, and constant maintenance costs, it is a very interesting alternative. This partial replication policy may be interesting for Globule, which currently does not have a decentralized replication policy.

# Bibliography

[1] M. Afergan, J. Wein, and A. LaMeyer, *Experience with some Principles for Building and Internet-Scale Reliable System*, In Proc. of the Workshop on Real, Large-Scale Distributed Systems, December 2005.

[2] S. Androutsellis-Theotokis and D. Spinellis, *A Survey of Peer-to-Peer Content Distribution Technologies*, ACM Computing Surveys 36(4), pp. 335-371, December 2004.

[3] M. Bhide, P. Deolasee, A. Katkar, A. Panchbudhe, K. Ramamritham and P. Shenoy, *Adaptive Push-Pull: Disseminating Dynamic Web data*. IEEE Transactions on Computers 51, 6 (June), pp. 652-668, 2002.

[4] P. Cao and C. Liu, *Maintaining Strong Cache Consistency in the World-Wide Web*, IEEE Transactions on Computers 47, 4 (April), pp. 445-457, 1998.

[5] V. Cate, *Alex - a Global Filesystem*, In Proc. File Systems Workshop (Ann Harbor, MI), USENIX, Berkeley, CA, pp. 1-11, 1992.

[6] Y. Chen, R. Katz and J. Kubiatowicz, *Dynamic Replica Placement for Scalable Content Delivery*, In Proc. 1st International Workshop on peer-to-Peer Systems (Cambridge, MA), Lecture Notes on Computer Science, vol. 2429. Springer-Verlag, Berlin, pp. 306-318, 2002.

[7] F. Dabek, M. Kaashoek, D. Karger, R. Morris, and I. Stoica, *Wide-area cooperative storage with CFS*, In Proc. ACM SOSP'01, Banff, Canada, pp. 202-215, 2001.

[8] Gnutella, http://gnutella.wego.com.

[9] M. Jelasity and O. Babaoglu, *T-Man: Fast Gossip-based Construction of Large-scale Overlay Topologies*, Technical Report UBLCS-2004-7, May 2004.

[10] M. Jelasity and M. van Steen, *Large-Scale Newscast Computing on the Internet*, Technical Report IR-503, Vrije Universiteit, Department of Computer Science, October 2002.

[11] J. Kangasharju, J. Roberts and K. Ross, *Object Replication Strategies in Content Distribution Networks*, In Proc. 6th Web Caching Workshop (Boston, MA), North-Holland, Amsterdam, 2001.

[12] KaZaa, http://www.kazaa.com.

[13] D. Karger, E. Lehman, T. Leighton, M. Levine, D. Lewin, and R. Panigrahy, *Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web*, In Proc. of the 29th Annual ACM Symposium on Theory of Computing, pp. 654-663, May 1997.

[14] Napster, http://www.napster.com.

[15] B. Neuman, *Scale in Distributed Systems*, Readings in Distributed Computing Systems, IEEE Computer Society Press, Los Alamitos, CA, pp. 463-489, 1994.

[16] G. Pierre and M. van Steen, *Globule: A Collaborative Content Delivery Network*, November 2005.

[17] L. Qiu, V. Padmanabhan and G. Voelker, *On the Placement of Web Server Replicas*, 2001.

[18] M. Rabinovich and A. Aggarwal, *RaDaR: A Scalable Architecture for a Global Web Hosting Service*, Computer Networks 31, 11-16, pp. 1545-1561, 1999.

[19] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, *A Scalable Content-Addressable Network*, In Proc. of SIGCOMM 2001, 2001.

[20] P. Rodriguez and S. Sibal, *SPREAD: Scalable Platform for Reliable and Efficient Automated Distribution*, Computer Networks 33, 1-6, pp. 33-46, 2000.

[21] A. Rowstron and P. Druschel, *Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems*, In Proc. of IFIP / ACM Middleware, Heidelberg, Germany, 2001.

[22] S. Sivasubramanian, M. Szymaniak, G. Pierre and M. van Steen, *Replication for Web Hosting Systems*, ACM Computing Surveys 36(3), September 2004.

[23] I. Stoica, R. Morris, D. Liben-Nowell, D. Karger, M. Kaashoek, F. Dabek, and H. Balakrishnan, *Chord: A Scalable Peer-to-peer Lookup Protocol for Internet Applications*, IEEE/ACM Trans. On Networking 11, pp. 17-32, February 2003.

[24] A. Tanenbaum and M. van Steen, *Distributed Systems Principles and Paradigms*, Prentice-Hall, 2002.

[25] F. Torres-Rojas, M. Ahamad and M. Raynal, *Timed Consistency for Shared Distributed Objects*, In Proc. 18th Symposium on Principles of Distributed Computing (Atlanta, GA), ACM Press, New York, NY, pp. 163-172, 1999.

[26] S. Voulgaris, D. Gavidia, and M. van Steen, *CYCLON, Inexpensive Membership Management for Unstructured P2P Overlays*, Journal of Network and Systems Management, Vol. 13, No. 2, June 2005.

[27] S. Voulgaris, M. Jelasity, and M. van Steen, *A Robust and Scalable Peer-to-Peer Gossiping Protocol*, In Agents and Peer-to-Peer Computing workshop, Melbourne, Australia, July 2003.

[28] S. Voulgaris, E. Rivière, A. Kermarrec, and M. van Steen, *Sub-2-Sub: Self-Organizing Content-Based Publish Subscribe for Dynamic Large Scale Collaborative Networks*, 5th Int'l Workshop on Peer-to-Peer Systems (IPTPS 2006), Santa Barbara, California, USA, February 2006.

[29] S. Voulgaris and M. van Steen, *Epidemic-style Management of Semantic Overlays for Content-Based Searching*, In Proc. EuroPar 2005, Lisbon, Portugal, August 2005.

[30] H. Yu and A. Vahdat, *Design and Evaluation of a Conit-Based Continuous Consistency Model for Replicated Services*, ACM Transactions on Computer Systems 20, 3, pp. 239-282, August 2002.