High Data Availability and Consistency for Distributed Hash Tables Deployed in Dynamic Peer-to-peer Communities

Vom Fachbereich 20 - Informatik
der Technischen Universität Darmstadt
zur Erlangung des akademischen Grades eines
Doktor-Ingenieurs (Dr.-Ing.)
genehmigte

Dissertation

von
Diplom-Ingenieur

Predrag Knežević

geboren in Zrenjanin, Serbien

Referent: Prof. Dr. Erich J. Neuhold
Korreferent: Prof. Dr. Karl Aberer

Tag der Einreichung: 24. April 2007
Tag der Mündlichen Prüfung: 10. Juli 2007

Darmstadt 2007
Hochschulkennziffer: D17

Acknowledgments

First and foremost, I would like to express my thanks to my academic supervisor Prof. Dr.-Ing. Erich J. Neuhold, who guided the research presented in this thesis to its very end. I am pretty sure that without his help it would be much harder to focus in given area, recognize unsolved issues, and propose proper solutions.

Special thanks are due to Prof. Dr. Karl Aberer. Apart from being my second adviser, he has credits for discovering Fraunhofer IPSI Institute to me and recommending it as a good place for research that could match to my interests. I am glad that I followed his advice. I would lack a very important experience, if I did not.

I am also like to thank to Prof. Dr. Alejandro Buchmann, Prof. Dr.-Ing. Ralf Steinmetz, and Prof. Dr. rer. nat. Max Mühlhäuser for being the member of the thesis committee and for the time and effort that they have invested in judging the contents of my thesis.

Among the colleagues, whom I worked with, I would like to express my gratitude first to Dr. Peter Fankhauser. As the head of the former OASYS division, he gave me the opportunity to join Fraunhofer IPSI Institute. Although we were focused on different research topics, our passion for drinking good espresso coffee and many internal colloquiums provided a good framework for many valuable discussions.

The road towards the final version of the thesis would be much harder without my colleagues from OASYS and later i-INFO division. I am immensely grateful to Dr. Andreas Wombacher and Dr. Thomas Risse for a countless number of fruitful discussions and their contributions in various phases of my research. Their help was invaluable in shaping the approach and bringing the text of the thesis into the present form. Although my German improved significantly during the last five years being here, *Deutsche Zusammenfassung* of the thesis needed a touch of a native speaker, and I thank Marco Paukert for his time and effort.

Being motivated to work hard on projects and PhD is hardly possible without strong support from friends and family, especially if you live abroad. Many IPSI colleagues became very close friends making to feel in Darmstadt as at home. On other hand, long distance puts old friendships on probe. Thus, I am really grateful to the old friends for all the moments that we had together, despite kilometers between us.

I will be always immensely grateful to my parents Nebojša and Verica and my sister Nataša for their everlasting love and suport ever since. Last but not least, my profound gratitude and love go to my wife Bojana and my son Damjan. Without their solid-steel love, unconditional support and understanding, everything would be much, much harder. This thesis is dedicated to them.

Abstract

Decentralized and peer-to-peer computing, as a subset of distributed computing, are seen as enabling technologies for future Internet applications. However, many of them require some sort of data management. Apart from currently popular P2P file-sharing, there are already application scenarios that need data management similar to existing distributed data management, but being deployable in highly dynamic environments.

Due to frequent changes of the peer availability, an essential issue in peer-to-peer data management is to keep data highly available and consistent with a very high probability. Usually, data are replicated at creation, hoping that at least one replica is available when needed. However, due to unpredictable behavior of peers, their availability varies and the number of configured replicas might not guarantee the intended data availability. Instead of fixing the number of replicas, the requested guarantees should be achieved by adapting the number of replicas at run-time in an autonomous way.

This thesis presents a decentralized and self-adaptable replication protocol that is able to guarantee high data availability and consistency fully transparently in a dynamic Distributed Hash Table. The proposed solution is generic and can be applied on the top of any DHT implementation that supports common DHT API. The protocol can detect a change in the peer availability and the replication factor will be adjusted according to the new settings, keeping or recovering the requested guarantees.

The protocol is based on two important assumptions: (1) ability to build and manage a decentralized replica directory and (2) ability to measure precisely the actual peer availability in the system. The replica directory is built on top of the DHT by using a key generation schema and wrapping replicas with additional system information such as version and replica ordinal number. The way in which replicas are managed in the DHT helps us to define a measurement technique for estimating peer availability. Peers cannot be checked directly, due to the fact that the common DHT API does not reveal any details about the underlying DHT topology. The peer availability is computed based on the measured the availability of replicas. With the help of confidence interval theory, it is possible to determine the sufficient number of probes that produces results with an acceptable error.

Finally, two variants of the protocol are defined: one that assumes that data are immutable, and another one without such a limitation. A theoretical model is developed to determine the sufficient number of replicas needed to deliver the pre-configured guarantees. If a peer detects that the current availability of peers and the replication factor are not sufficient for maintaining the guarantees, the sufficient replication factor will be determined according to the measured availability of peers. Knowing the previous and new replication factor, the peer is able to insert into the DHT additional replicas of data managed in its local storage. On the other hand, if the number of replicas is higher than needed, the peer will remove unnecessary replicas from its storage, reducing the storage overhead.

Replication makes the consistency of data harder to maintain. Every logical update is translated into a set of replica updates. Due to the dynamic nature of the DHT, many replicas can be

unavailable at the time when an application issues an update request. Such conditions force the usage of some weak-consistency models that updates all available replicas and synchronizes all the others eventually when they become online again. Until this is achieved, none of guarantees about the consistency of the accessed data cannot be given. The proposed protocol implements a weak-consistency mechanism that, unlike the others, is able to provide an arbitrary high probabilistic guarantees about the consistency of available data before all replicas are synchronized. This is done by updating the available and inserting the new version of all offline replicas. As soon as a replica becomes available again, it is informed about missing updates, and is merged with the new version. Such approach ensures that at least one consistent replica is available with a high probability when data are requested.

The approach presented was evaluated by using a custom-made simulator. The requested availability and consistency levels are fully guaranteed in the DHT with a stable or increasing peer availability. During churns (periods when the peer availability decreases), the guarantees are maintained only in cases when the dynamic of churns is low (the peer availability decreases slowly). Faster changes cannot be compensated fully, but eventually, after the system stabilizes enough replicas will be generated, and the guarantees will be recovered.

Deutsche Zusammenfassung

Dezentralisierte Systeme und peer-to-peer (P2P) Computing werden als technologische Voraussetzungen zukünftiger Internetanwendungen gesehen. Obwohl sich die Infrastrukturen unterscheiden, sind die Aufgaben der Anwendungen gleich oder ähnlich zu denjenigen Anwendungen, die den klassischen Client/Server-Ansatz benutzen: Bearbeitung und Speicherung von Daten. Abgesehen vom derzeitig populären P2P File-sharing gibt es viele Anwendungsszenarios, die eine Datenverwaltung benötigen, die ähnlich zu einer verteilten Datenverwaltung ist. Gleichzeitig muss sie aber in hoch-dynamischen Umgebungen funktionieren.

Das größte Problem der P2P-Datenverwaltung ist die Gewährleistung und Konsistenzhaltung der Daten. Üblicherweise werden Daten bei der Erstellung repliziert und man hofft, dass mindestens eine der Repliken verfügbar ist, wenn sie gebraucht wird. Aufgrund unvorhersehbaren Verhalten der Peers garantiert die konfigurierte Anzahl der Repliken oft nicht die gewünschte Datenverfügbarkeit. Statt einer vordefinierten Anzahl von Repliken sollte sich die Anzahl der Repliken während der Laufzeit voll-autonom an die aktuelle Peerverfügbarkeit anpassen.

Diese Doktorarbeit präsentiert ein dezentralisiertes und selbst-adaptives Replikationsprotokoll, das voll-transparent eine hohe Datenverfügbarkeit und -konsistenz mittels einer dynamischen verteilten Hash-Tabelle (DHT – Distributed Hash Tables auf Englisch) garantiert. Die Lösung ist voll generisch und auf alle DHT-Implementierungen anwendbar, die DHT API bieten. Wenn sich die Peerverfügbarkeit ändert, dann muss das Protokoll in der Lage sein, dies festzustellen. Auf Basis der aktuellen Werte wird der Replikationsfaktor verändert, um die gewünschte Verfügbarkeit und Konsistenz zu behalten.

Das Replikationsprotokoll basiert auf zwei wichtigen Aufnahmen: (1) ein Vermögen um ein dezentralisiertes Replikenverzeichnis aufzubauen und zu pflegen, und (2) ein Vermögen um genau die aktuelle Peerverfügbarkeit im System zu messen. Das Replikenverzeichnis benutzt die verteilte Hash-Tabelle als Basis und generiert mit der Hilfe einer definierten Funktion Schlüssel. Repliken sind mit zusätzlichen Systeminformationen wie ihre Version und Ordnungszahl unter den Schlüsseln gespeichert. Der Ansatz macht ermöglicht es, später eine Messmethode für die Peerverfügbarkeit zu definieren. Weil das gemeine DHT API keine Information über die Systemtopologie publiziert, ist es nicht möglich die Verfügbarkeit von den Peers direkt zu prüfen. Stattdessen prüft die vorgeschlagene Messmethode die Verfügbarkeit der Repliken. Es ist möglich mit der Hilfe der Konfidenzintervalltheorie zu berechnen, wie viele Proben ausreichen, dass die Messgebnisse einen niedrigen Fehler haben.

Schließlich werden zwei Varianten des Protokolls definiert: Die erste nimmt an, dass Daten unveränderlich bleiben. Die zweite erlaubt, dass Daten während der Laufzeit modifiziert werden dürfen. Um festzulegen wie viele Repliken für gewisse konfigurierte Garantien gebraucht werden, wird ein analytisches Modell entwickelt. Wenn ein Peer feststellt, dass die aktuelle Peerverfügbarkeit und die Anzahl der Repliken nicht die gewünschte Garantien liefern, wird ein neuer Replikationsfaktor berechnet. Jetzt weiß der Peer, wie viele zusätzliche Repliken der Daten eingefügt werden sollen. Andererseits löschen Peers unnötige Repliken aus ihren Speichern, wenn die gleichen Garantien mit weniger Repliken auch erreicht werden.

Datenreplikation erschwert die Erhaltung der Datenkonsistenz. Jede logische Datenaktualisierung wird in der Aktualisierung von mehreren Repliken übersetzt. Viele dieser Repliken können nicht zu jedem Zeitpunkt gefunden werden, da Peers, bei denen die Repliken gespeichert sind, nicht verfügbar sein können. Solche Bedingungen erlauben nur, dass die Daten eine schwache Konsistenz haben, bis alle Repliken aktualisiert sind. Vor der Aktualisierung kann keine Aussage über die Konsistenz der gefunden Daten gemacht werden. Das Replikationsprotokoll dieser Arbeit implementiert ein schwach-konsistentes Modell, aber mit einem Unterschied zu den bestehenden: Es bietet eine hohe Wahrscheinlichkeit, dass die gefundenen Daten, die vor allen Repliken aktualisiert werden, konsistent sind. So etwas ist möglich, wenn alle verfügbaren Repliken aktualisiert werden und dazu eine neue Version aller Offline-Repliken eingefügt werden. Sobald diese Repliken wieder online sind, werden über verpassende Updates informiert werden, werden die alten und neuen Versionen vereinigt. Deswegen garantiert das Protokoll mit einer hohen Wahrscheinlichkeit, dass mindestens eine konsistente Replik verfügbar ist.

Der Ansatz wird mit Hilfe eines selbst-entwickeltes Simulators evaluiert. Die Ergebnisse zeigen, dass die Datenverfügbarkeit und -konsistenz voll-garantiert sind, wenn die Peerverfügbarkeit stabil oder ansteigend ist. Bei absteigender Verfügbarkeit, werden die Garantien nur bei langsamen Änderungen voll gepflegt. Bei schneller Senkung der Peerverfügbarkeit fallen die Garantien für einen gewissen Zeitabschnitt, aber nach der Stabilisierung des Systems werden wieder genug Repliken generiert, um die gewünschte Datenverfügbarkeit und -konsistenz wieder zu bekommen.

Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

WITH the expansion of the Internet, many database systems have moved to the new infrastructure and introduced new issues to existing data management. Communication between clients and servers is not as reliable as on local networks and communication links are usually under control of third parties. Servers become overloaded easily if the database system is not scalable enough. Even distributed database systems (DDBS) suffer from similar problems, because they are designed to operate in stable, controlled environments.

Service oriented and peer-to-peer computing are seen as an enabling technology for future Internet applications in different areas, e.g. e-Business and e-Science. In the long run, such an approach should deliver better scalability and performance than solutions based on the classical client-server architecture. An advantage of an application built on top of a P2P architecture lies in a better usage of all available resources that are spread among all participating machines, including users' desktops and laptops. On other hand, application logic should not need to concern itself with resource or data distribution, i.e. they should be fully transparent.

As discussed in [RKW04], many application scenarios wish to profit from the advantages of a decentralized/P2P infrastructure, and at the same time, they require data management that is more sophisticated than just plain data sharing. It should be as reliable as those in the client-server or the classical distributed environments. For example, service-oriented computing relies on the existence of a service directory, containing various kinds of service information used during the service discovery phase. State-of-the-art solutions place the directory on a dedicated machine. Building a workflow from the existing services requires finding them in the service directory. If the workflow design is based on top of a decentralized/P2P infrastructure, the service directory should be decentralized as well, but its functionalities should remain unchanged. Service descriptions should be spread among participating peers, but access to them must be fully transparent for the services. It should be possible to add new peers,update and search the existing description without knowing anything about the underlying infrastructure, nor resource and data distribution.

Fairly often services combined into workflows need storage that will keep temporary data used at different stages of the workflow execution. In a decentralized environment, storage must be decentralized as well. However, the data management provided must have properties similar to those guaranteed by centralized solutions.

Unfortunately, decentralized data management cannot be realized by using existing database solutions, because they can operate only in the presence of a controlled environment. Hence, P2P/decentralized applications need a decentralized datastore that could be deployed in highly dynamic peer-to-peer networks and at the same time, it must be reliable, respect privacy and ensure data security.

Existing peer-to-peer applications like PAST [RD01b], Skype [1], or OceanStore [KBC+00] use Distributed Hash Tables (DHTs) as a basis for managing their data. They are essentially overlay networks that provide deterministic routing and at the same time deliver good scalability in terms of network delays (the number of hops to reach the requested destination peer) and the number of messages generated . However, making data management fully reliable is not an easy task. It is achieved by guaranteeing the availability and consistency of the managed data, and having the ability to recover the system from network partitioning or machine failure.

Reliability is the subject of long-term research into standard distributed databases, and many of its aspects are well understood and solved. Since decentralized/peer-to-peer data management is a subclass of distributed data management, existing decentralized/P2P storages like PAST or OceanStore try to apply classical solutions in the new environment. To make them work, the new environment should behave similarly to those for which solutions have already been invented. Therefore, the above-mentioned P2P storages work well only if peers are highly available, their behavior is predictable or well known, and no network partitions occur. On the other hand, many applications must be deployed in an environment where such assumptions cannot hold. As a recent study [SGG02a, BSV03] shows, P2P networks are highly dynamic in reality. Peers are fully autonomous and can disappear from the network at any time, without any previous notification. Their availability is fairly low (around 20%), not known in advance, nor predictable, and can vary over time. Reliable data management on top of such a network cannot be realized using standard techniques known from distributed data management.

The research presented in this thesis has been motivated by the requirements defined within the BRICKS project, presented in detail in Chapter 2. Since it is fully decentralized, its aim is to integrate content from various sources and to provide transparent access to it, no matter where the data are actually managed . Even if many functionalities, e.g. distributed query processing, can be implemented in a decentralized way, there are always situations in which at least some sort of highly available storage is necessary. In BRICKS the need arises of having service descriptions, administrative information about collections, ontologies and some annotations globally available for all nodes. However, data managed by the decentralized storage should not disappear if some peers become unavailable, i.e. a high data availability must be guaranteed. At the same time, managed data are subject to update. Thus, the consistency of data

[1] http://www.skype.com

must be guaranteed as well.

This thesis focuses on guaranteeing an arbitrary, but highly available consistency of data managed in a DHT with an arbitrarily high peer offline/online rate. The required guarantees are given as input parameters to participating peers that deliver and maintain the guarantees over time. The replication protocol presented in the thesis is fully decentralized and can be applied to a DHT with an arbitrary peer availability, without knowing it in advance. Peers can leave or return online at any time without prior notification. A DHT equipped with our protocol preserves its set of functions, and extends them with transparent data availability and consistency guarantees. The logic of an application implemented on top becomes much simpler, because it does not have to worry about availability and consistency any more.

1.1 Distributed Hash Tables

A well-known approach for managing data in decentralized/peer-to-peer communities is using DHT (Distributed Hash Table) overlay networks. They emerged as an answer to performance problems that appeared in the first pure P2P systems such as Gnutella [Gnu01], where every request was broadcast to a node's neighbors. DHTs are low-level P2P systems that provide a consistent way of routing information to the final destination, can handle changes in topologies well and provide functions similar to a hash table data structure.

As suggested in [DZDS03], a DHT should provide at least the following functions:

- *route(Key, Message)* - routes a message deterministically to its final destination based on the supplied key
- *store(Key, Value)* - stores the value under the given key in the DHT and informs the initiator of the request if the operation has been successful or not
- *lookup(Key)* - reads from the DHT the value associated with the key and returns it to the initiator of the request

Both *store* and *lookup* are realized by using the *route* function. A *lookup* is initiated by routing a *LookupRequest* message with the given key. The peer that is responsible for the key responds with a *LookupResponse* message. Similarly, the *store* function is based on sending *StoreRequest* and receiving *StoreResponse* messages.

The main property of all DHTs is the deterministic routing of messages submitted by applications. Every peer is responsible for a portion of the key space nearest to its ID. The distance function can be defined in many ways, but usually it is a simple arithmetic distance between a peer ID and a key. Whenever a peer issues a *route*, *store* or *lookup* with a key, the message will end up on the peer responsible for that key.

We are going to demonstrate routing on Pastry [RD01a] – a DHT implementation where requests are routed towards their destination by using the following rules:

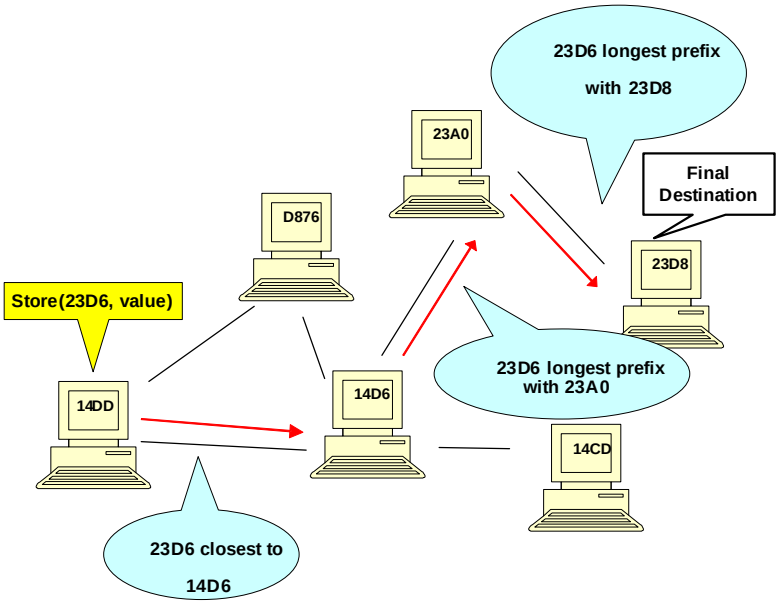


Figure 1.1: Storing a value in Pastry DHT

- Forward request to a peer whose ID has the longest common prefix (according to the hexadecimal representation) with the given *Key*, among all known neighbors
- Forward request to a peer, whose ID is the nearest to the given Key. The distance function is defined as the absolute value of numerical difference between ID and *Key*.

More about Pastry deterministic routing and the proof of its correctness can be found in [RD01a]. To demonstrate it on an example, Figure 1.1 shows a Pastry network of 6 peers, whose IDs are generated randomly. Their values are displayed on the stylized screens. A peer knows only about peers connected to it with solid lines. Let peer **14DD** decide to store a value under key **23D6**. The peer issues *store(23D6, value)*. The appropriate *StoreRequest* message is forwarded to peer **14D6**, because its ID is the closest to key **23D6**. Peer **14D6** checks where to forward the message further. It finds peer **23A0**, whose ID has the longest common prefix (2 digits) with the supplied key. By using the same rule, the request is forwarded further to peer **23D8**. This peer is the final destination for the message associated with key **23D6**, because the peer's ID is the nearest to it. Upon receipt, the peer processes the message, and stores the value in its local storage under the given key.

During a period of system stability (i.e. no changes in network topology), all messages with the same key will end up on the same destination peer, no matter which peer initiates the routing. If, for example, peer **14CD** (Figure 1.2) wants to access the previously stored value, it issues

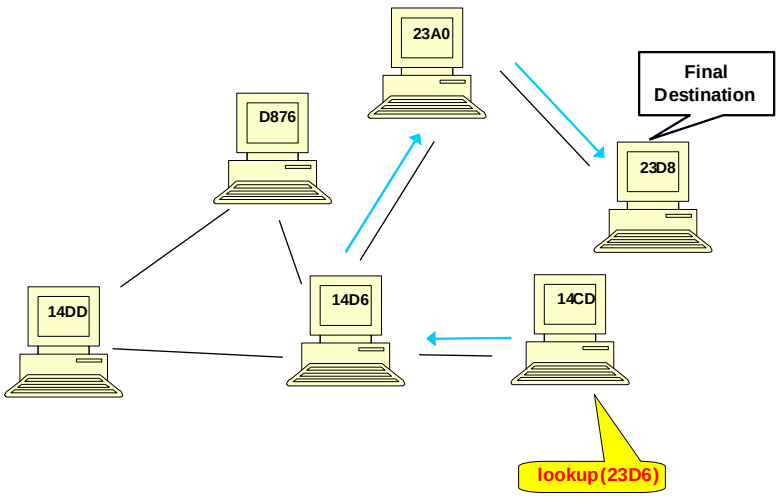


Figure 1.2: Accessing a stored value in Pastry DHT

lookup(23D6). Using the same routing rules, the *LookupRequest* message is forwarded first to peer **14D6**, and then via peer **23A0** to the final destination. Peer **23D8** processes the request, finds the value in its storage, and sends it back to the originator of the request.

When a peer goes offline, some other online peers now become responsible for the part of the keyspace that belonged to the offline peer. It will be split among peers that are currently nearest to keys in that part of the keyspace. On other hand, peers joining the DHT or coming back online will take responsibility for the part of the keyspace that has been under control of other peers until that moment. Figure 1.3 demonstrate the situation when peer **23D8** goes offline. All requests following *lookup(23D6)* will end up on peer **23A0**, because in the new topology its ID is the nearest one to the supplied key. The peer does not have the value in its storage, and therefore *lookups* will not be successful, as long as peer **23D8** remains offline.

When a peer rejoins the community, it can keep its old ID, and as a corollary the peer will be now responsible for a part of the keyspace that intersects at least with the previously managed part of keyspace.

When a peer want to join the community for the first time, its ID is usually generated at random and the peer joins only if the generated ID has not already been taken by somebody else. If already in use, the peer searches for another ID, until a free one is found.

DHT does not forbid the storage of more objects under the same key. Such situations can happen due to topology changes: a previously stored object can be offline, or the routing rules forward messages according to the current topology to another destination peer. However, among all objects stored under the same key, only one is accessible at any given moment, due to the deterministic routing rules. If peer **D876** issues a *store(23D6, value)* request in the DHT pre-

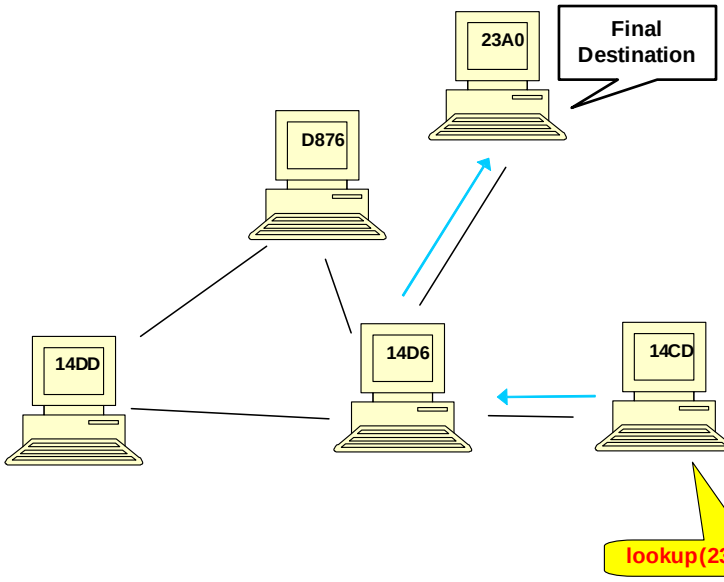


Figure 1.3: Topology change in Pastry DHT: peer **23D8** goes offline

sented in Figure 1.3, it ends up on peer **23A0**. The value will be stored, although the same or different value has been inserted already under the same key at peer **23D8** before. This cannot be detected, and therefore *store(23D6, value)* is successful. Until peer **23D8** stays offline, all following *lookup(23D6)* will be able to locate the value in DHT.

When peer **23D8** rejoins, the DHT topology reverts to the one presented in Figure 1.2. Now both peer **23A0** and **23D8** have a value under key **23D6** in their storages, but only the one managed by peer **23D8** is accessible. According to the current topology, all requests following *lookup(23D6)* will end up there.

Peer **23A0** is able to detect that key **23D6** does not belong to its part of the keyspace any longer. Two options are possible: to remove the key and the associated value from the storage, or to send them to a peer responsible for the key at that given moment. Finding the right peer is done simply by issuing *store(23D6, value)*. The value will be forwarded to the currently responsible peer. If it is managed there already, it will be overwritten. Otherwise, the value will be inserted. However, if the overwritten value is the latest one, the consistency of available data cannot be guaranteed. This can cause serious issues for many applications.

1.2 Problem Description

As mentioned earlier, due to their good qualities, many peer-to-peer applications build the management of their data on top of a DHT. In order to provide good quality of service to users,

applications demand from a storage a kind of quality of service, like guaranteed data availability, consistency, security or response time. If data are not available, many system functions will be unavailable as well, or outcomes will be unpredictable. Additionally, available data must be highly consistent, i.e. up-to-date, in order to enable proper system behavior, and to deliver the correct results to users.

1.2.1 Data Availability

As demonstrated in the previous Section (Figure 1.3), a plain DHT does not guarantee the availability of managed data. Whenever a peer goes offline, locally stored (*Key*, *Value*) pairs become inaccessible until the peer appears online again. In the example shown, as long as peer **23D8** is offline, data stored under the key **23D6** will remain unavailable.

Data availability can be achieved easily by replicating data a number of times, and storing the replicas on different peers in the community. In order to track their locations, a replica directory is used. As long as at least one peer which has a replica is online, data are available as well. Staying with our example, data stored under key **23D6** could be stored within the DHT several times under different keys: **23D6**, **D8A0** and **2300**. They all will be managed by different peers: **23D8**, **D876**, and **23A0** respectively. Having that, the data will be available for as long as at least one of the peers is online. Clearly, determining the correct replication factor depends on the requested data and the actual peer availability. To cope with these issues and deliver high data availability, solutions used in existing decentralized storages like PAST [RD01b], CFS [DKK+01], OceanStore [KBC+00] assume that the peer availability is high, highly predictable, and quite stable over time. Therefore, they are able to ensure high data availability with a fixed replication factor.

This assumption does not really hold in practice. The peer availability depends on user habits, time of day, and/or popularity of available content. Studies carried out [BSV03, SGG02a] confirm this and show that the actual peer availability is fairly low – around 20%. If we want to maintain data availability at a requested level all the time, the replication factor must be calculated on the basis of the minimum of peer availability. Doing it in advance is a hard problem, because peer's behavior is hard to predict. If we overestimate the peer availability, the system will never deliver the requested data availability. On the other hand, if we underestimate it, the system will generate unnecessary high costs. Setting the right replication factor would require shutting down the DHT, changing the replication factor at every peer, and starting the DHT again. This is hardly possible in reality, because peers are fully autonomous and not under the control of a single authority.

The real challenge would be to develop a mechanism that self-adapts the replication factor based on the actual peer availability and the requested data availability.

1.2.2 Data Consistency

Replication easily increases availability. On other hand, it makes consistency harder to maintain if data are mutable. Consistency can be forced by choosing a strict consistency model for the managed data, where data can be updated only if all replicas are available at that moment. As already mentioned, peer availability is fairly low in reality. Thus, applying strict consistency would limit significantly the possibility to update data, which is unacceptable for many applications. Choosing a weak consistency model allows updates even when some replicas are offline. In such a case there is a risk that they become available at some moment later. They are going to be synchronized eventually, but meanwhile accessing them can give the false impression that the update did not happen. Applications that need to get the current data would behave unpredictably if consistency is not ensured.

In the previously used example, we have demonstrated that availability of data can be increased by creating more replicas and storing them under different keys. As mentioned above, a weak consistency model allows updates even if all replicas are not available at that moment. It is enough to have at least one of the replicas managed under keys **23D6**, **D8A0** and **2300** online in order to perform an update. However, at any subsequent moment, the up-to-date replicas can disappear and incorrect replicas can again be available. If an application finds the incorrect replica, it will have no idea that some updates were performed in the past, and therefore it could start to behave unpredictably.

A disadvantage of the existing weak consistency models is that they do not provide any probabilistic guarantees on the consistency of available data. Until all replicas are synchronized, consistency can only be forced by allowing access only to up-to-date replicas limiting the availability of data. This is not usually acceptable, and therefore another alternative is to allow access to any available replicas. Existing solutions do not give any probabilistic guarantees on the consistency of available data, nor on the time needed to bring all replicas up-to-date.

Since updates should address all available replicas, another potential source of inconsistency is concurrent updates. It is essential to define a sort of coordination that ensures the update order of all available replicas. Without it, concurrent updates might leave the replicas in an unpredictable state - they could have a value provided by any of the individual updates. The standard distributed data management handles concurrent updates via dedicated machines or by using master-slave/primary-secondary replica concepts. This works only under an assumption that the coordination machine, the master or the primary replica are highly available. Existing decentralized/P2P storages like OceanStore [KBC+00] adopt the same approach, assuming that some peers/replicas are almost always present in the system.

1.2.3 Challenges

Keeping the high availability and maintaining the consistency of data in dynamic DHTs poses the following challenges:

- Self-adapting the replication factor based on the actual peer availability and the requested

data availability

- The consistency of updated data should be guaranteed with the requested probability
- Defining a fully decentralized mechanism for coordinating concurrent updates.

1.2.4 Goals

This thesis should provide a solution for the challenges presented. The designed replication protocol should reach, maintain, and recover the requested guarantees under the following conditions:

- Requested data availability and consistency guarantees are specified at deployment time
- Guarantees must be provided in a fully transparent way
- The number of peers in a DHT and their IDs/addresses are unknown
- Peer IDs are generated randomly and are unique
- Peers are fully autonomous and behave similarly, i.e. can be described with the same peer availability p
- Peers can go offline at any time without prior notification
- After coming online again,a peer keeps its previously assigned ID and stored data
- The number of managed objects in a DHT is unknown
- Data access pattern is unknown $\Rightarrow$ all managed data must have the same availability and consistency
- Replicas of an object are stored on different peers
- Peers do not need to have synchronized internal clocks

1.3 Contributions

Designing and implementing a replication protocol that solves the challenges requires solutions to the following subproblems:

- Defining a decentralized replica directory

  Replication protocols require access to a directory that maintains information about replicas and their locations. Standard centralized directories cannot be deployed in a decentralized environment. Thus, the solution presented in this thesis is based on a key generation

schema, used for generating keys under which replicas of an object are stored and retrieved to/from a DHT. All replica keys are globally unique, but replica keys of an object are correlated, i.e. by knowing only the object key, all its replica keys can be computed by every peer locally.

- Precise estimation of the actual peer availability at run-time

  Changes in peer availability have a crucial impact on the availability of data. Keeping guarantees on the requested level is possible only if the actual peer availability is known. The measurement method is inspired by confidence interval theory. The mechanism is not aware of the underlying DHT topology, and therefore, peer availability is estimated indirectly by measuring replica availability. A peer generates a number of replica keys out of the replicas managed locally, and checks if the chosen replicas are online by using available DHT functions. To obtain an acceptable error, a limited number of peers are polled for their probing results, and the final measurement value is computed as an aggregation of the collected values. The proposed measurement method is fully decentralized, and besides the estimation of the actual peer availability in the system, the availability trends can be obtained as well.

- Guaranteeing the pre-configured data availability and consistency by adjusting the number of replicas of

  – immutable data

    A theoretical model is developed to determine the sufficient number of replicas which are required to guarantee the pre-configured data availability. If a peer detects that data should be replicated more times than before, it will create additional replicas of the data managed locally. On the other hand, if the number of replicas is higher than needed, the peer will remove unnecessary replicas,thereby reducing the storage overhead. High data availability is fully guaranteed when peer availability is stable or increased, and unneeded replicas are eventually removed. During churns, data availability can be guaranteed fully only in cases when churn rates are low, i.e. the peer availability in the system decreases slowly. Stronger churns cannot be compensated fully immediately, but eventually, after the peer availability stabilizes, enough replicas will be generated, and the requested data availability will be recovered.

  – mutable data

    The proposed replication mechanism is extended with a weak consistency model. Unlike the existing solutions, it is able to provide guarantees on the consistency of updated data, until all replicas are eventually synchronized. This is achieved by updating all available replicas, and inserting the new version of offline replicas in the system. When an offline replica comes back online, synchronization will be triggered by the peer holding the new version. The number of replicas that should

be addressed by every update does not differ from the case when the guarantees are supposed to be delivered on immutable data. Again, it depends only on the requested guarantees and the actual peer availability and is fully independent of the distribution of the updates applied.

1.4 Outline

The rest of this thesis is structured as follows. Chapter 2 gives details of the application scenario that motivated the research presented in this thesis. The goals and the architecture of the system is presented, arguing why the peer-to-peer approach was chosen, and why the application requires data management that should guarantee high availability and consistency of data.

Chapter 3 presents related work. According to the definition given, decentralized/peer-to-peer data management belongs to the field of distributed data management. Therefore, problems similar to those addressed in this thesis are well researched in the area of standard distributed databases, and the most important solutions are presented in this chapter. Later on, the current situation in decentralized/P2P data management is presented: which of the existing solutions can be reused and where the limitations are. All methods presented are illustrated with systems that implement them.

Chapter 4 presents in detail our replication protocol. It is fully decentralized and based on functions provided by the underlying DHT. The chapter starts first by defining a decentralized replica directory. It is essential for the further design of various protocol mechanisms. For example, replica location should be known for measuring actual peer availability, adding new replicas, or finding the replica of an object.

Afterwards, we define a replication protocol that delivers the requested availability of immutable data, no matter what the initial replication factor was. DHT functions are redefined to deliver the guarantees transparently. Analysis of the protocol introduced clearly shows that the number of replicas depends on the requested data availability and the actual peer availability.

In order to support management of mutable data, the previous protocol is extended to allow updates. The supported consistency model belongs to the weak consistency group. Unlike state-of-the-art solutions, we provide an arbitrary consistency guarantee until all replicas are eventually updated. As for immutable data, the analysis shows that the number of replicas needed depends on the requested consistency and availability level, and the actual peer availability.

The proposed protocol can be made self-adaptable if peers could measure the actual peer availability. Based on the value obtained , they could adjust the replication factor accordingly. Hence, the measurements must be precise. The proposed technique uses confidence interval theory to determine the sufficient number of probes for the requested precision. Further, to ensure good precision during churns, the current peer availability is calculated via a regression curve fitted to the measurements.

Both protocol variants are evaluated in detail in Chapter 5. The defined tests check if the goals are fulfilled. The results obtained confirm that the designed protocol is able to reach, maintain

and recover the requested availability and consistency of data managed in a DHT. At the same time, storage overhead is kept at the predicted level.

Finally, Chapter 6 summarizes the contributions of the thesis. Based on some limitations discovered in the approach presented and issues not covered by this work, potential future research directions are proposed.

Chapter 2

BRICKS Project

THIS Chapter provides details of the European project BRICKS[1] [BRI04], whose application scenario has motivated the research carried out in this thesis. The aim of the project is to enable integrated access to distributed resources in the Cultural Heritage domain. The target audience is very broad and heterogeneous and involves cultural heritage and educational institutions, the research community, industry, and the general public. The project idea is motivated by the fact that the amount of digital information and digitized content is continuously increasing but still much effort has to be expended to discover and access it. The reasons for such a situation are heterogeneous data formats, restricted access, proprietary access interfaces, etc. Typical usage scenarios are integrated queries among several knowledge resource, e.g. to discover all Italian artifacts from the Renaissance in European museums. Another example is to follow the life cycle of historic documents, whose physical copies are distributed all over Europe.

A standard method for integrated access is to place all available content and metadata in a central place. Unfortunately, such a solution requires a quite powerful and costly infrastructure if the volume of data is large. Considerations of cost optimization are highly important for Cultural Heritage institutions, especially if they are funded from public money. Therefore, better usage of the existing resources, i.e. a decentralized/P2P approach promises to deliver a significantly less costly system,and does not mean sacrificing too much on the performance side.

The following Section presents the requirements that the BRICKS architecture must fulfill. The architecture itself is described in Section 2.2

2.1 Requirements

Besides better usage of existing resources, the decentralized and service-oriented approach has an additional advantage: all its functionalities are spread over many locations, ensuring that the

[1] BRICKS - Building Resources for Integrated Cultural Knowledge Services (IST 507457)

system can survive a number of failures while being able to continue to be operable. If a server crashes in the client-server architecture, the whole system stops until the server is repaired. The reliability of servers could be increased by replicating their functionalities in the system, but this would increase the total of system maintenance required.

It is always a good strategy to design a system that is able to handle loads which were not foreseen during the initial design phase. For standard client-server the system load must be carefully estimated, e.g. by guessing the maximum number of users. In open systems like BRICKS such estimation is hardly possible. Due to the simplicity of joining the system the number of nodes can increase rapidly. Thanks to better usage of available resources, loads should be distributed within the system without losing much on performance. Thus, the decentralized approach ensures that the system will perform well even when given unpredictable loads.

The lack of central points removes the need for centralized system maintenance. Important infrastructure functionalities are all implemented in a self-organizing way. That is a strong advantage of the decentralized approach because a centralized administration costs additional money and personnel must be dedicated to the tasks. Keeping the cost of the system maintenance at a minimum is one of the major requirements, and the decentralized approach is a way to achieve it.

In order to reduce the costs further, the system should be deployed on top of the regular Internet. With its expansion in the last few years, connectivity can be obtained anywhere at a reasonable low cost. Institutions should be able to reuse their existing Internet connection when joining the BRICKS community.

Figure 2.1 shows a possible topology of the future BRICKS system. Every node represents a member institution, where the software for accessing BRICKS is installed. Such nodes are called BNodes. BNodes communicate with each other and use available resources for content and metadata management.

To summarize, the BRICKS infrastructure should be fully decentralized, use the Internet as a communication medium and fulfill the following requirements:

- Expandability – the ability to acquire new services, new content, or new users, without any interruption of service.
- Scalability – the ability to maintain excellence in service quality, as the volumes of requests, of content and of users increase.
- Availability – the ability to operate in a reliable way over the longest possible time interval.
- Graduality of Engagement – the ability to offer a wide spectrum of solutions to the content and service providers that want to become members of BRICKS.
- Interoperability – the ability to make available services to and exploit services from other digital libraries.

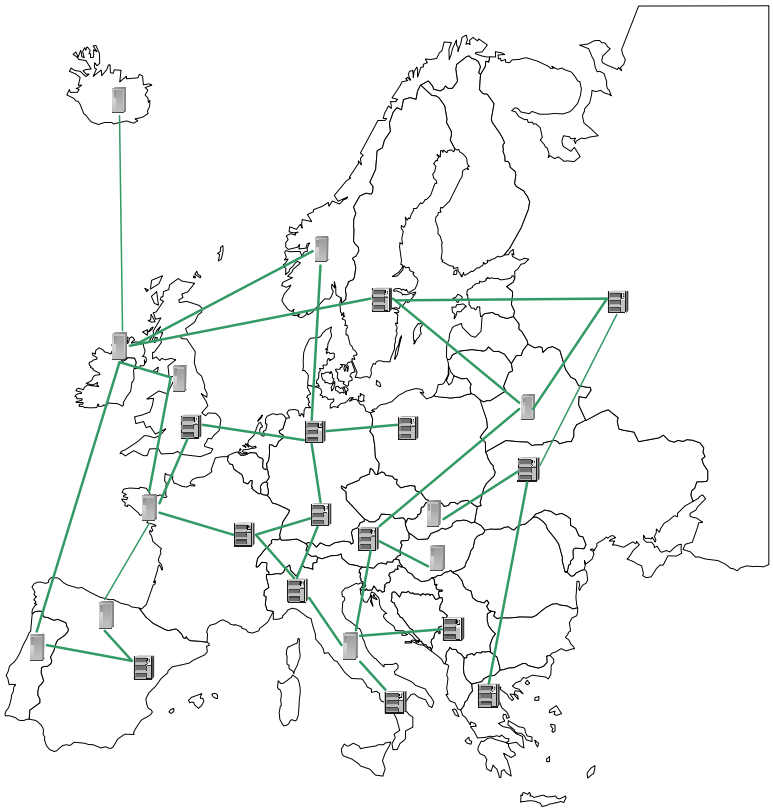


Figure 2.1: Decentralized BRICKS topology

- Low-cost – an institution can become a BRICKS member with minimal investment. In the ideal case the institution should only get the BRICKS software distribution, which will be available free of charge, install it, connect it to the Internet and become a BRICKS member. To minimize the maintenance cost of the infrastructure any central organization, which maintains e.g. the service directory, should be avoided. Instead, the infrastructure should be self-organizing so that the scalability and availability of the fundamental services, e.g. service discovery or metadata storage, are guaranteed.

- Flexible Membership – institutions can join or leave the system at any point in time without significant administrative overheads.

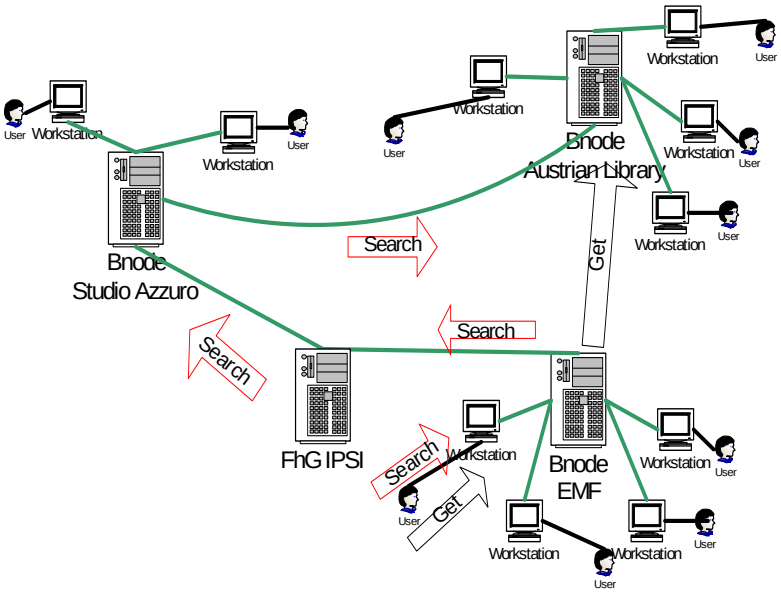


Figure 2.2: Request routing in BRICKS

2.2 Approach

The major challenge in designing a fully decentralized system like BRICKS is to fulfill the listed requirements without the help of a central coordination unit. Every BNode knows directly only a subset of other BNodes in the system. However, if a BNode wants to reach another member that is unknown to it, it will forward a request to some of its known neighbors that will deliver the request to the final destination or forward it again. This is depicted in Figure 2.2. It shows also that BRICKS users access the system only through a local BNode available at their institution. Hence every user request is first sent to the institution's BNode and then the request is routed between other BNodes to the final destination. Search requests behave like that; the BNode will preselect a list of BNodes where a search request could be fulfilled, and then the BNode will route it there. When the location of the content is known, e.g. as a result of the query, the BNode will directly be contacted.

2.2.1 BNode

A BNode could be seen as a set of services that are required to manage an institution's presence in the system, and to provide services for the rest of the community. They run within a Web service framework that provides a standard set of functionalities: service management/invocation, parameter serialization/deserialization. The architecture (component-based view) is shown in

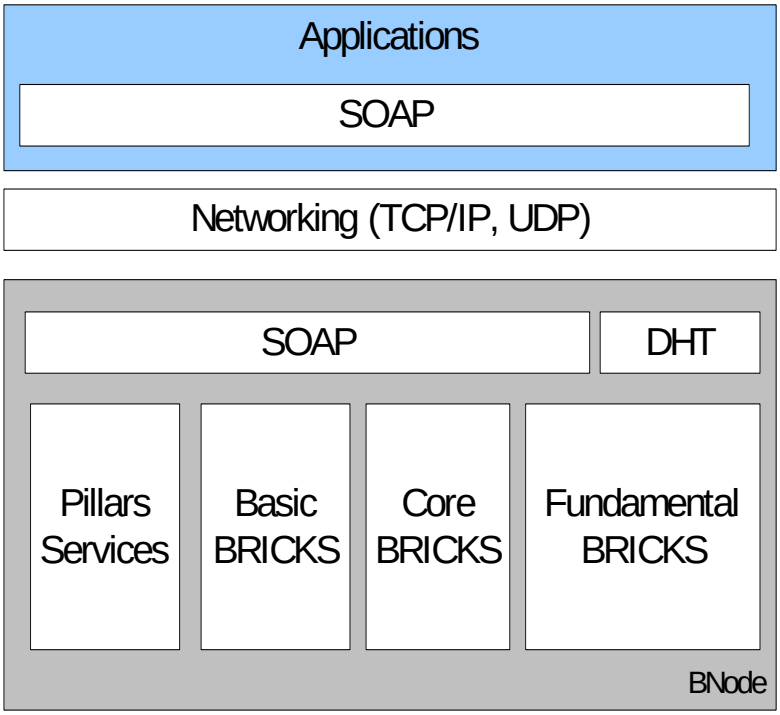


Figure 2.3: BNode architecture (component-based view)

Figure 2.3. In order to present how applications communicate with BNode, the networking layer is placed between them. Application requests and responses are sent and received using SOAP.

The BNode consists of three types of components: fundamental, core and basic Bricks. Most of them are standard Web services, described by WSDL (Web Services Description Language)documents, and registered with a UDDI (Universal Description, Discovery and Integration) compatible repository used also for discovering appropriate services. Since the BNode architecture is service-based, a BNode installation can be spread over more than one machine at the installation site. In such a case, fundamental Bricks (Figure 2.4) are needed on every machine that is part of the local installation, and core and basic Bricks could be present only on some machines. As their name suggests, core Bricks provide core system functionalities to users, i.e. a minimal set of services that enables users to use the system. On other hand, basic Bricks are optional, and they need not be present at every installation site.

Each BNode organizes it own content into collections which are later exposed to the rest of the community. However, in order to make them really visible, collection metadata should be managed by a storage that is accessible for all BNodes in a transparent way. In classic client-

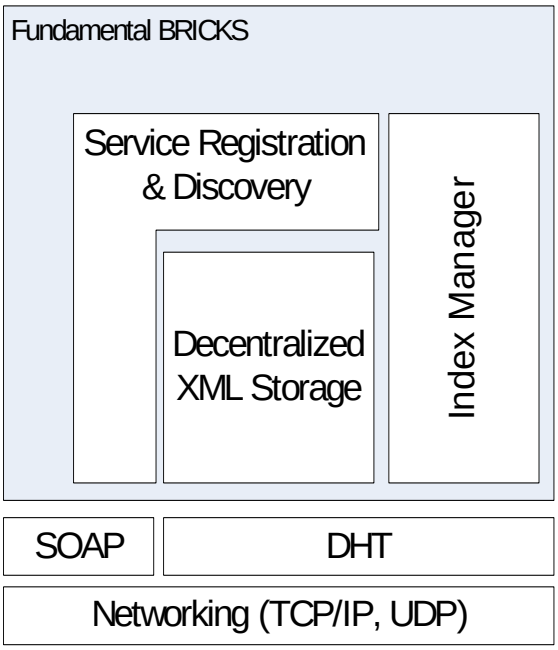


Figure 2.4: Fundamental BRICKS (layered view)

server or distributed system such storage is placed on a dedicated machine that is known to all BNodes in the community. Unfortunately, this is in conflict with the decentralized approach taken by the BRICKS project. The storage itself must be decentralized as well, and at the same its usage must be transparent and not much different from the classical one.

The storage can be used for managing other system related data as well. For example, descriptions of all deployed services can be stored there, and used by all BNodes in the service discovery phase. At the moment, web services are described by using WSDL and published with the help of UDDI in service directories, which is one sort of centralized database. Since our design does not allow for a centralized directory, its functionalities should be implemented on top of the decentralized storage. As presented in Figure 2.4, service registration and discovery in BRICKS is based on top of our decentralized storage.

Since WSDL service descriptions are XML documents and the collection metadata can be defined using XML as well, the proposed decentralized storage could work natively with XML. The XML data model is essentially a tree and it is used for representing semistructured and hierarchical data, and as data exchange format in heterogeneous systems with many different schemas. The distribution and partitioning schemas of the XML data model are more versatile compared to the relational data model, and at the same time a tree can be maintained more easily than a graph.

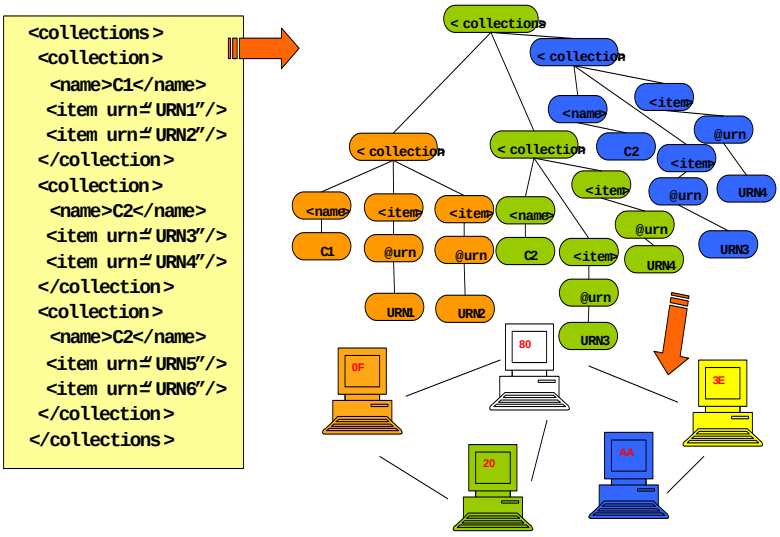


Figure 2.5: Managing an XML document on the top of the DHT

2.2.2 Decentralized XML Storage

The approach taken differs from existing solutions, like Gnutella [Gnu01] or KaZaA [KaZ02], where each peer makes their local files available for community download. In our approach XML documents are split into finer pieces that are then spread in the DHT. The documents are created and modified by the peers at run-time and can be accessed from any peer in a uniform way, e.g. a peer does not have to know anything about data location. Figure 2.5 shows how XML documents are managed with a DHT. The example demonstrates that every XML document (on the left) can be represented as a tree. Afterwards, individual tree nodes or complete subtrees are inserted into the DHT, keeping the tree structure intact. Later on, by knowing the reference to the root of the document, it is possible to access, browse and search the stored document.

The proposed storage implements the Document Object Model (DOM) [W3C02] interface, which has been widely adopted among developers, and a significant legacy of applications is already built on top of the DOM. Therefore, many applications can be ported easily to the new environment. XML query languages such as XPath [W3C99] or XQuery [W3C03] can use the DOM as well, so they could be used for querying of the datastore. Finer data granularity will make querying and updating more efficient.

The storage must be able to operate in highly dynamic communities. Peers can depart or join at any time, nobody has a global system overview nor can rely on any particular peer. These requirements differ significantly from those in distributed databases where node leaving is only due to some node failure and the system overview is globally known.

Figure 2.6 presents the proposed XML storage architecture. All layers exist on every peer in

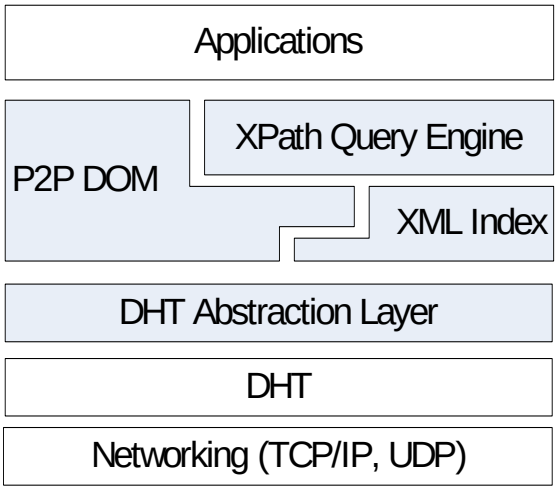


Figure 2.6: Decentralized XML Storage

the system. The storage is accessed through the P2P-DOM component or by using the XPath query engine. The query engine could be supported by an optional index manager which maintains indices in the DHT as well. P2P-DOM exports a large portion of the DOM [W3C02] interface to applications, and maintains a part of a XML tree in a local store (e.g. files, database). It serves local requests (adding, updating and removing of DOM-tree nodes) and requests coming from other peers through the DHT overlay, and tries to keep the decentralized database in a consistent state. In order to make the DHT layer pluggable, it is wrapped in a layer that unifies APIs of particular implementations, so the upper layer does not need to be modified.

Designing the decentralized XML storage faces several important challenges, as stated in [Kne04]:

Data partitioning As Figure 2.5 might suggest, XML documents can be partitioned in many different ways. Two extreme cases are managing each XML node separately, or all nodes together (XML documents are not partitioned). Between these extremes, nodes could be grouped in an arbitrary way and managed together. The partitioning policy applied makes a great impact on the XML storage access performance, and the real challenge is to define a policy that delivers good performance for the most common application scenarios.

Data placement Increasing the storage performance further requires choosing an appropriate data placement policy. Data should be managed close to peers that need them often, or such that they can be cached locally.

Concurrency control Since many peers could modify an XML document concurrently, a concurrency control mechanism is of utmost importance for synchronizing individual modifications to an XML document. None of the concurrent updates should leave the storage

in unpredictable and inconsistent state.

Reliability It is a mixture of several properties. The storage must:

- Ensure high availability and consistency of managed data
- Show resistance to network partitioning, and peer departure/arrivals

Reliability and concurrency control are highly generic challenges, fully independent from the XML data model. The XML storage architecture in Figure 2.6 shows that they could be solved within the DHT layer. Since the DHT does not guarantee the availability nor consistency of managed data natively, the guarantees should be achieved by introducing a replication protocol. It should deliver an arbitrarily high availability and consistency of data in a highly dynamic DHT. These challenges represent the main focus of this thesis.

2.3 Summary

This chapter has given an overview of the BRICKS project, in particular its requirements and the proposed decentralized architecture. The application scenario presented requires a decentralized XML storage that provides the same guarantees as classical centralized storage: data must be highly available and consistent. Also, data distribution must be fully hidden, i.e. their location must be fully transparent for applications.

The proposed XML storage manages its data within a DHT. DHT lacks data availability and consistency guarantees natively due to peer's unpredictable behavior. Therefore, DHT should be extended with a replication protocol, which will provide them.

Chapter 3

Related Work

DATA management and how to make it reliable on unreliable hardware have been the subject of research for many years. Although hardware reliability is increasing constantly, failure probability is still too high for applications that need 100% reliability. Thus, distributed data management is seen as a solution. However, just storing data redundantly does not bring reliability automatically. A mechanism that ensures data consistency, and recovery from failures and network partitioning must be provided. Furthermore, distributed data management must be:

- Transparent
- Reliable
- Performant
- Scalable

The following section categorizes distributed architectures and shows how peer-to-peer approaches (in general, and especially Distributed Hash Tables) are related. Section 3.3 presents existing approaches for dealing with data availability and consistency in the standard distributed data management. Later on, Section 3.4 summarizes the state of the art in decentralized/P2P data management. It shows which of the techniques from the standard distributed data management are still applicable, which modifications are necessary, and where the limitations are. Finally, we compare them with our approach, and point out differences.

3.1 Distributed Architectures

As defined in [CD89], a distributed system is "one in which components located at networked computers communicate and coordinate their actions only by passing messages". Such a defini-

tion matches a broad spectrum of systems, from traditional clusters, through distributed databases and filesystems, and to decentralized and peer-to-peer systems.

Further analyzing the definition, one can see three degrees of freedom, by which such distributed systems can be additionally classified:

- Availability of components
- Communication pattern among components
- Quality of communication links

Particular instances of the listed classification define an architecture of a distributed system:

- Tightly coupled (clustered) – a set of highly available machines connected via very fast, fault-tolerant links. The machines run the same process in parallel, subdividing the task in parts that are made individually by each one, and then put back together to make the final result
- Client-server – the client machine contacts the highly-available server for data, then formats and displays them to the user. Also, the client commits data to the server if they should be permanently changed. Clients do not exchange messages directly; all communication goes via the server. The quality of communication links can vary from fast, reliable local area networks (LANs) to less reliable wide area networks (WANs) or the Internet. Every machine/node has a global system overview, i.e. it knows where the other machines are, their addresses, services, etc. Also, it should be part of the system most of the time. Going offline and disconnections are considered as crashes that will be eventually repaired.
- 3-tier architecture – it moves the client logic to a middle tier so that stateless clients can be used. The communication pattern is the same as in the client-server architecture, and such an architecture can be deployed on a wide variety of networks as well.
- N-tier architecture – further generalization of the 3-tier architecture. It refers typically to web applications which forward their requests to other enterprise services
- Peer-to-peer – an architecture where there are no special machines that provide dedicated services or manage network resources. Instead all responsibilities are distributed among all machines, known as peers. The availability of peers is much lower than in the previously described architectures. Such an architecture is usually deployed on the Internet, where network links are usually under the control of third parties, connection speeds and bandwidths can vary and the links themselves can often be down. Peers are completely autonomous; they join and leave the system at any point in time, i.e. are totally unpredictable. Going offline is not a fault or crash, and can happen quite often. The other peers should be aware of it and have methods for compensation. No one has a global view of the system, i.e. peers are aware of some parts of the system topology.

3.2 Peer-to-peer Architectures

Although not really new, peer-to-peer (P2P) architectures have become very popular in the last few years with the increase of processing power of regular desktop machines and improved bandwidth that connects them to the Internet. Essentially, the first Internet services like DNS [AL01] and Usenet [HA87] were peer-to-peer in their nature. In the very early days of the Internet, the number of available machines was rather low, and their performance was fairly similar. Therefore, the peer-to-peer architecture was quite an obvious choice. With the expansion of personal (desktop) computers, the processing power of available machines became quite unbalanced: plenty of not-so-powerful PCs, and a significantly lower number of powerful mainframes/servers. The client-server approach was therefore the natural choice for that period of time. Again, in recent years, PCs have been getting more and more powerful, matching and leaving behind the power of old mainframes and servers.

The resources available nowadays allow us to perform many tasks previously executed on the server side. Also, the world-wide availability of applications deployed on top of the Internet potentially attracts a large number of users. A classical client-server architecture does not scale well if the number of clients/users increases significantly. If this happens, a typical solution would be to put more resources (bandwidth, processing power, memory, or storage) on the server side, hoping that such a new configuration can cope with users' requests for a longer period. Making a client-server architecture fault-tolerant requires even more investments on the server side: replicating resources even at geographically distant locations, and this is usually quite costly.

The P2P approach is seen as a solution for reducing a system's costs and improving its scalability and reliability. Milojičić et al. [MKL+02] present an excellent survey of P2P computing, i.e. existing architectures, application areas, and challenges. Depending on their degree of decentralization, we can distinguish between the following architectures:

- Hybrid – only some system functions are decentralized. For example, data are transferred directly among peers, whereas authentication, authorization, and search are performed on a server.
- Super-peer – highly available, powerful peers (super-peers) form a network that takes over the role of a server. The rest of the peers act as clients of such a super-peer network.
- Pure – all system functions are decentralized where every peer provides the same functions potentially to all other peers.

Pure systems can be further classified according to communication pattern between a peer and its neighbors:

- unstructured – a peer sends requests to all its neighbors, which forwards them to their neighbors, until the request reaches its final destination. An obvious disadvantage is high

bandwidth consumption. On the other hand, the approach requires minimal knowledge about the rest of the peers. Hence, maintaining membership is cheap. However, the lack of routing rules cannot ensure that a request will ever end up on a peer that could return a proper answer.

- structured – the clear disadvantages of the unstructured approach motivated the research community to come up with solutions that are able to reduce bandwidth usage by maintaining routing rules. Instead of broadcasting, requests are routed deterministically towards their destinations.

Plaxton et al. [PRR97] first demonstrated the idea of deterministic routing within a static distributed environment. The approach was used for efficiently accessing copies of object. Data and peers are associated with different IDs. They are location-independent and before applying any operation, an object has to be found in the network. Therefore, a proper request is routed through the network using a simple rule: in every hop the request is forwarded towards a peer whose ID has the longest common prefix/suffix with the given object ID. The paper proves that such a routing is deterministic. Also, it shows that the number of hops needed follows $O(\log N)$ complexity, where N is the number of peers in the network.

The routing mechanism presented was not truly self-adaptable. It assumed a static distributed environments, where peers do not leave, i.e. the system topology does not change. However, Plaxton et al. provided a basis for future research in structured P2P systems. They are also known as Distributed Hash Tables (DHT), because they provide the semantics of hash table data structure on top of a network of peers.

The most interesting representatives of DHTs are Pastry [RD01a], Tapestry [ZKJ01], P-Grid [Abe01], CAN [RFH$^+$01], Viceroy [MNR02], Kademlia [MM02], and Chord [SMK$^+$01]. They all have similar complexity, and differ mostly in handling peer's joining/departure, and building/propagating routing rules.

For example, Chord places all objects and peer IDs in a virtual circle, and defines that an object belongs to a peer if the object ID is equal to or follows the peer ID on the virtual circle. Knowing this, requests are then routed along the circle until a proper peer is reached. P-Grid takes a somewhat different approach – it organizes keys in a virtual tree divided into a set of subtrees managed by the peers. Since all keys in a particular subtree share the same key prefix, every request should be routed towards a peer responsible for a subtree that shares the largest common prefix with the given key.

As we are going to see later in the chapter, due to their good properties, DHTs are used as a basis for more complex P2P systems. Unfortunately, none of the existing DHTs guarantees the availability of managed data. When a peer goes offline, data stored locally in its storage become unavailable as well.

Backx et al. [BWDD02] give a good comparison of the different P2P architectures with respect to real implementations. Obviously unstructured system do not scale well in terms of traffic, and therefore Backx suggests that system architects should consider, if application requirements allow ,super-peer architectures to be a reasonable trade-off. The paper did not take

into account Distributed Hash Tables as an alternative. The super-peer approach is further evaluated by Yang and Garcia-Molina [YGM03], who provide practical guidelines about designing a performant and reliable super-peer network.

3.3 Distributed Data Management

Advantages of distributed data management were recognized very early [Com75]. Increased data availability and performances were key motivating factors for the larger research community and industry. During the 60s and 70s, the available communication infrastructure provided much lower bandwidths than now, and it was obvious that system performance could be improved by copying/replicating data closer to their consumers.

3.3.1 Availability

The availability of data can be increased by managing them redundantly. An obvious way for achieving this is by replicating data on several machines, hoping that at least one copy will be available when the data are requested. If an object is replicated on R different machines, under an assumption that their availability is equal, the availability a of the object becomes:

$$a = 1 - (1 - p)^R$$

where p is availability of machines where a replica is stored. In order to keep data distribution transparent to applications, a replica directory is needed. It tracks location of all replicas of managed objects, and is usually managed on a dedicated machine. To be able to access data, the rest of the nodes must know its location. The directory itself could be further replicated to increase its fault-tolerance.

An alternative way for increasing data availability is by using erasure coding. Namely, data of m blocks length are recoded into n blocks length, where $n > m$. Due to added redundancy, reconstructing the original data can be done from any of the new m blocks. Assuming that blocks are stored on different machines, data availability can be expressed then as:

$$a = \sum_{i=m}^{n} \left(\begin{array}{c} n \\ i \end{array} \right) p^i (1 - p)^{n-i}$$

Originally, erasure codes come from the computer communication field, where they are used for error correction and establishing reliable communication channels. They were not considered for a long time as a solution for increasing data availability. Communication channels were for a long time expensive and their speed was poor. Under such conditions, erasure codes had a serious drawback: every data access requires finding and fetching m blocks from the network, and reconstructing the original data. If communication channels are slow, data access becomes extremely inefficient. Also, data reconstruction needs some processing power, and it could be a potential bottleneck if the data access rate is high.

Therefore, first distributed databases like SDD-1 [BSJBR80], INGRES [SN77], and POREL [NB77] used replication in order to ensure data availability.

On the other hand, as pointed out by Lin et al. [LCL04], if machine availability is higher than 50% (usually the case), erasure coding provides lower storage overhead for the same data availability. The first usage of erasure coding for increasing data availability happened within a computer itself, where the connectivity is much faster and cheaper. The approach formally defined as RAID by Patterson et al. [PGK88] chunks data transparently into a higher number of blocks and places them on multiple hard disks. Access performances are increased, because data are reconstructed from the blocks read in parallel. At the same time, the availability of data is significantly increased, and the system can survive even multiple hard disk crashes.

The emergence of fast Local Area Networks (LANs) brought the first network file systems. The Network File System [SGK+85] and the Andrew File System [Cam97] replicate files or their portions on client machines to obtain better performance. However, files on NFS servers are not replicated on more machines in order to keep availability high. It is assumed that an NFS server is highly available, and equipped with RAID.

Other proposals like xFS [ADN+96] are serverless, i.e. it distributes file server processing responsibilities across a set of available computers in a local network at the granularity of individual files. In order to deal with failures and to provide high availability, xFS applies erasure coding and implements a software RAID storage system by striping files across many machines.

3.3.2 Consistency

Replication increases data availability easily, but makes consistency harder to maintain, if data can be modified at run-time. If a replica is unreachable during an update (machine crash or network partitioning), its value cannot be changed. Therefore, this replica should not be used for read access until it has been synchronized with the latest value.

A good survey of existing update and recovery methods has been written by Ceri et al. [CHKS91] and Chen and Pu [CP92]. There are essentially two groups of replication protocols: strict, that enforce one-copy equivalence as the correctness criterion, and weak, that allow different replica versions in the system.

ROWA (Read One Write All) [BHG87] is a strict protocol; it converts a logical read to a read of any of the replicas, and converts a logical write to a write of all the replicas. It is simple and elegant, but has a significant drawback. If any of the replicas is unavailable, a write operation cannot be performed.

Its slight modification is ROWA-A (Read One Write All Available), where write operations are executed on all available copies. A variation of ROWA-A has been proposed by Bernstein et al. [BG84], where the coordinator of an update transaction sends each write operation to all the sites where replicas reside and wait for confirmation. If a timeout occurs before acknowledgment from all sites, it assumes that those which have not replied are unavailable and continues with the update on the available sites. The unavailable sites will update their copies to the latest state when they recover. In the case of a network partitioning, updates are still allowed, and this may

lead potentially to data inconsistency. Such a situation can be overcome by introducing a notion of "primary partition". If replicas from the primary partition are not available, neither reads nor writes on an object are allowed. Additional discussion about other ROWA-A variants can be found in [HBH96].

Another class of the strict replication protocols are those based on voting and quorums [Tho79]. Originally, Thomas proposed that each read and write operation has to obtain a sufficient number of votes (i.e. quorum) to be able to commit. Quorum-based protocols work well under network partitioning, i.e. they ensure that an update cannot happen if enough replicas are not available. Unfortunately, they obtain quorum for reading the data as well, which makes it more expensive in term of generated traffic.

Quorums are quite often suggested as a way to reduce the overall overhead of replication. They can deal with network partitions, minimize communication overhead, increase availability, and balance the cost between read and write operations [CD89, BHG87]. The recent research done by Jiménez-Peris et al [JPPMAK03] studied the most relevant forms of quorums and compares them with the conventional ROWA-A. Surprisingly, the study showed that ROWA-A delivers performance at least as good as the best quorum approach. Most of the time, ROWA-A outperforms them, and is much easier to implement in practice.

Weak consistency schemes allow multiple data versions to exist in the system at the same time, but all old versions will be updated eventually. The master-slave class of protocols distinguishes between master and slave copies; a master copy is always updated first, and then the master propagates the changes to slave replicas. It does not need to happen immediately. The slave's notification can be delayed, in order to optimize communication costs. If recovery is needed, slave copies must synchronize with the master after coming online again, or re-establishing network connectivity. For example, Stonebraker proposed the "primary site" model [Sto79] for distributed INGRES. Namely, he distinguishes primary from secondary replicas. All updates are first sent to primary replicas. Afterwards, primary replicas update slaves. Distributed INGRES tends to ensure that 95% of traffic is local (during the 70s, communication links were slow and expensive). The database is not fully replicated at every site, but if an object is not found at the local site, it will be replicated for all later accesses. Secondary replicas will be created, and primary replicas are informed about this.

As already mentioned, a large threat to data consistency is network partitioning - a situation when one or more communication links are down, and as a consequence, the distributed system is divided into two or more isolated partitions. Depending on the replication protocol used, some data might become inconsistent if they are updated. When the network is repaired, inconsistent data should be recovered to a consistent state. An excellent survey about keeping and recovering data consistency in partitioned networks was presented by Davidson et al. [DGMS85]. Based on how replication protocols behave in partitioned networks, we distinguish between two approaches: pessimistic and optimistic. Pessimistic protocols prevent inconsistency by limiting availability. They make the worst-case assumptions about how often partitions occur. Since consistency is preserved, partition merging is followed by straightforward data synchronization: updates done during partitioning (if any) are propagated to non-updated data copies. On the other

hand, optimistic protocols do not limit availability. Updates may be executed in any partition as long as it contains copies of requested data. The main assumption here is that the partitioning rarely occurs. However, after partitions are merged, the system must first detect inconsistencies and resolve them. A proper merging can be done only if the semantics of conflicted data is known.

From the previously introduced replication protocols, all except ROWA-A belong to the pessimistic group. ROWA-A allows updates until at least a single data copy is available, but after reconnection any consistency must be resolved manually. An optimistic replication protocol proposed by Davidson [Dav84] uses a precedence graph to detect inconsistencies. It is constructed from logs that record the order of reads and writes on data items. The graph is constructed after reconnection in order to check if the data item is inconsistent or not. If the graph obtained is acyclic, and even if the data item has been updated in more than one partition, the updates can be merged into a new consistent state. A cyclic graph signals conflicts – they are resolved by rolling back until the resulting precedence graph is acyclic.

3.3.3 Placement

Although stated as one of the advantages of distributed data management, data replication does not always guarantee an improvement in system performances, i.e. a decrease in response time. As pointed out by Garcia-Molina and Barbara [GHB81], besides obvious storage costs, replication introduces additional communication costs generated by a more complicated concurrency control and commit mechanism.

Getting better performance for reasonable costs requires that the replication protocols take into account replica placement strategies as well. The optimal strategy strongly depends on communication link capacity, application location, application access pattern, and network topology. For example, an application that almost always reads, and seldom writes would benefit most from a fully replicated database/storage. At the other extreme, an application that often writes should experience better performance if fewer replicas are present in a system.

Finding the optimal placement in an unconstrained environment is an NP-complete problem, as pointed out by Eswaran [Esw74]. Therefore, Ceri et al. [CMP82] define a strategy based on access-cost minimization, i.e. the system constraints such as the available storage space, cost of local and remote access, the number of files, read/write frequency. In order to compute the optimal solution, the given constraints must be known during the system design phase.

In many cases, constraints cannot be estimated well, and therefore it would be better if the replica placement strategy adapted at run-time. Wolfson et al. [WJH97] proposed an adaptive data replication (ADR) mechanism, which changes the replica placement strategy according to the data read-write pattern. The mechanism works on top of the ROWA protocol, and may be combined with two-phase locking or another concurrency control algorithm. The aim of ADR is to decrease communication costs for data access. It is convergent-optimal. Objects that are frequently read are replicated more and moved closer to machines that need them. In contrast, object that are updated often drop some replicas and the rest of them are moved to machines

with good communication links.

In order to ensure data availability without knowing a machine's fault probability, Pu et al. [PNP88] suggested that missing replicas should be regenerated. The mechanism is based on the ROWA protocol, with a slight modification: if a write operation cannot address some replicas, they will be re-created.

3.4 Decentralized/Peer-to-peer Data Management

Although peers can be unavailable often due to their owners' decision, or an unreliable Internet connection, decentralized/P2P data management should still have the same properties as standard distributed data management.

3.4.1 Availability

Bhagwan et al. [BMSV02] present an overview of existing techniques and provide an analytical model for reasoning about the efficiency/costs of replication and erasure coding in a peer-to-peer environment. Their analysis showed that erasure coding enables the requested availability with a significantly lower storage overhead. For example, if peer availability is 50% on average, maintaining 99% data availability requires approximately 10 replicas, whereas erasure coding produces only 2.5 times higher storage overhead. The drawbacks of erasure coding is that every data access must involve a number of peers that own the data blocks needed. The blocks must be transferred over the network to a peer that requests them, and then the original data are reconstructed. Some peers could be on a slow network connection therefore making access to some data much slower, because the reader cannot reconstruct the original data until all the blocks needed have arrived. Another disadvantage of erasure coding is a higher sensitivity to a peer's offline rate. If the average peer availability decreases, the data availability obtained decreases much faster if data redundancy is achieved by using erasure coding.

A similar comparison of two approaches was made by Weatherspoon and Kubiatowicz [WK02]. They tested replication versus erasure coding on a set of highly available servers that build a decentralized storage. Such a scenario benefits fully if erasure coding is applied. However, the authors were aware of its weaknesses, and suggested a sort of intelligent buffering and caching on the client side to overcome communication delays.

In order to see what a potential storage overhead for real peer-to-peer systems is, Bhagwan et al. [BSV03] traced peer availability in the Overnet [Wik03b] network, and Saroiu et al. [SGG02a] performed a similar experiment for Napster [Nap01] and Gnutella [Gnu01] networks. The majority of peers in all observed networks have an availability of around 20%. Under such a condition, erasure coding is no longer the preferred solution. As pointed out by Lin et al. [LCL04] and what was not considered by the previous authors, if the peer availability is low (between 20% - 50%), replication might provide a higher data availability than the erasure coding keeping the storage overhead at the same level.

In addition, Rodrigues and Liskov [RL05] argue that the usage of erasure coding is questionable even if the peer availability is higher. Their results are obtained using the traces from PlanetLab [Pla06], OverNet, and Farsite [Far02]. The authors confirmed that erasure coding provides a lower storage overhead. However, maintaining a highly available storage generates high traffic (around 100kbs per peer), and therefore makes the solution inappropriate for networks with limited bandwidths. On the other hand, using erasure coding introduces an additional complexity to the system. Not only is there complexity associated with the encoding and decoding of the blocks, but the entire system design becomes more complex. For example, recovering data availability requires the generation of missing blocks, but in order to do that, all the data must be reconstructed first on a peer.

No matter how redundancy is built into the system, the key issue is how to determine the right replication/stretch factor for making data available under certain guarantees. It depends strongly on the peer availability. Predicting peers' behavior is a hard task, and is often even impossible. Therefore, most approaches are built on the assumption that peer availability is high and stable, or does not fall below a given threshold. However, if the assumption does not hold, data availability might not be guaranteed at all. The system should be stopped, reconfigured with a new replication/stretch factor, and started again. Since peers are usually fully autonomous, and often unreachable, such a reconfiguration is practically impossible.

The approach presented in this thesis is self-configurable. The requested data availability is supplied to peers at deployment phase. At run-time peers periodically measure the actual peer availability and adjust the number of replicas of managed data so that the requested data availability is achieved and maintained.

Cuenca-Acuna et al. [CAMN03] recognized that the replication/stretch factor is hard to predict. Therefore, they try to achieve high file availability at run-time. The approach relies upon a global, centralized index that holds information about files' availability. Before storing, files are recoded using Red Solomon codes (a variant of the erasure coding) and chunked into blocks distributed among peers. Peers subsequently check on a random basis the availability of files whose blocks are in its storage. If the availability is below a given threshold, the blocks are replicated and sent to some random peers. Such a replication schema makes file updates impossible, because it is not known how to find all blocks. Also, the approach depends on a centralized index, whereas our protocol is fully decentralized.

3.4.2 Consistency

As we have seen, keeping redundant data consistent is not a simple issue even in standard distributed data management, where a machine goes offline only due to a failure, and network partitioning happens rarely.

Due to the potentially low availability of peers, the strict consistency model (one copy serializability) is not suitable for P2P data management. It would definitely ensure data consistency, but would drastically limit the number of possible updates, because almost always at least one of the replicas is not available. Thus, P2P data management can be based on one of the following

models:

- No consistency model
- Weak consistency model
- Continuous consistency model

No Consistency Model

The lack of a consistency model poses a limitation if consistency should be guaranteed: data cannot be updated after storing. However, this is not an issue for a wide range of applications like data sharing and archival systems.

Data-sharing Applications

P2P attracted wider attention with the appearance of Napster [Nap01] – an application for music sharing on the Internet. Its architecture is hybrid and usually known as the first generation of P2P file-sharing applications. Authentication, authorization and search were performed on a server, and once proper files are found – they were transferred directly between peers. Napster's appearance showed that the P2P approach has a great potential in terms of cost reduction and scalability.

A common usage scenario for all P2P data sharing system follows a very simple pattern. Namely, users, after getting membership, expose their files to others and at the same time are able to search through all other available files. The query semantics are quite limited. A user can narrow the search by requesting that some keywords appear within titles, select some switches based on file type, length, or user defined category. If the search is successful, the user has a chance to contact peers with interesting files and copy them. Afterwards, the copied files become available for others as well. Usually, all files with the same name appear independently in a result list.

Very soon, people recognized Napster's big disadvantage, which also made it very efficient. It relied on a central server that indexed all files available in the community. In case of a failure, it would be extremely easy to stop the whole network. As a response, Gnutella [Gnu01] appeared in early 2000. Initial popularity of the network was spurred on by Napster's threatened legal demise in early 2001. It did not have any central index, all queries were propagated to peer's neighbors, and the neighborhood forwarded them further until matches were found and sent back to the query originator. Having the result list, the user could select some files as before, and copy them directly from peers that own them.

Obviously, doing search via flooding did not deliver good performance as noticed by Ritter [Rit01] and many others. Gnutella was considered as unscalable, and inspired the development of Distributed Hash Tables. To address scalability problems, the creators of Gnutella switched to super-peer architecture, where super-peers (ultrapeers in Gnutella terminology) are responsible for request routing, and regular peers (leaves in Gnutella terminology) forward their

requests to ultrapeers only. These modifications improved Gnutella performance significantly, as pointed out in [Bab06].

The disadvantages of Napster were recognized by many others, but they saw the danger of going in the fully decentralized direction. KaZaA [KaZ02] was designed on top of the FastTrack protocol, whose details were never published officially, but quite a lot of information was obtained through reverse engineering [Wik06]. Powerful machines with enough bandwidth are promoted to super-peers that perform indexing and manage a part of a global index. A really new feature of the FastTrack protocol was the facility to increase file download speed by contacting multiple peers, and getting different portions of a file simultaneously.

In 2004, KaZaA's popularity was overtaken by the eDonkey [Wik03a] network. It added two more features: the ability to detect the same content in the network even if published using different filenames, and already completed pieces of a file being downloaded could be offered to others, so the overall download time is usually much lower. The eDonkey network is based on super-peers, where they are called servers. Anybody with enough resources can add a new server to the network. A disadvantage is that the whole network depends on them. In order to overcome such a dependency, the authors[1] designed a successor – OverNet [Wik03b] that is said to use a variant of Kademlia DHT [MM02] for managing the global index.

Dating back to 2001, the BitTorrent [Bit06, Tho05] protocol's popularity has been increasing rapidly in the last few years. Bram Cohen designed it, including lessons learned from the previously described P2P file-sharing networks. The protocol itself does not include search support, due to potential copyright issues. It is a scalable solution for distributing large files by spreading the bandwidth load among all peers that already have some portion of the requested file. Similar to eDonkey, while a file is being downloaded, it is already available for the other peers as well. Thus, downloading files can be done by getting pieces from different peers and combining them together locally. Since peers are usually on different parts of the Internet, bandwidth utilization will be much higher, i.e. transfer time lower. The protocol performances depend heavily on a peer's wish to share already downloaded content with the others. Unfortunately, many users are not actually interested in sharing already downloaded content. Such behavior is called free riding, and Adar and Huberman performed a good analysis for the existing Gnutella networks [AH00]. Therefore, BitTorrent measures the upload/download ratio for every client, and those with a higher ratio get higher download speeds that allow download of complete files much faster. A download starts by reading a *torrent* file that contains some file metadata and the location of the tracker responsible for the given torrent. The peer then contacts the tracker, getting information about peers that are downloading the file at that moment. Afterwards, the peer is ready to start trading file pieces and choose appropriate peers in order to get the maximum download rate. Due to its excellent performances, BitTorrent attracts a growing number of organizations and individuals that are using it for distributing legal materials. Since trackers are not a part of the protocol, access to them can be restricted according to various policies.

To summarize, the popular P2P file-sharing systems presented do not have any built-in support

[1] MetaMachine Inc. (`http://www.metamachine.com`)

for managing high availability and consistency of shared data. Downloading a file can be seen as its replication, and indeed its availability increases afterwards. Consequently, the availability of popular files is pretty high, whereas rarely requested content suffers from low availability. However, as already confirmed in practice, this works fine for sharing multimedia content.

Such an implicit replication scheme makes it impossible to add any sort of update mechanism. None of the peers knows how many times a particular file is replicated and where the copies are. Therefore, a new version of a file must be published separately, even using a new filename. Peers that have an old file version cannot detect that the new version is available and update their local copy. The P2P file-sharing networks are perfectly suited for content distribution scenarios, but not for any other sort of more sophisticated data management.

Archival Systems

FreeNet [Fre01, CSWH01, CMH+02] is an attempt to build a fully autonomous and decentralized storage. All files and complete communication are encrypted, so that users' identities or files' content can be kept secret. Every peer and file are associated with an ID. File reading and inserting requests are routed towards a peer with an ID closest to the given file ID. Similar to any P2P file-sharing system, file replication is rather implicit, and file availability is increased when a peer wants to access it. Namely, if the peer did not access file before, it is copied to a peer's local storage, and reused for all following operations. Also, if the same file is requested by other peers, and the peer receives such a request, it can directly answer, without the need to forward the request further. To summarize, FreeNet is mainly write-once storage. To keep average file availability at a high level, peers participating in FreeNet should be online for a large portion of time.

The Free Haven project [Fre03, DFM01] has similar aims. It is designed as a community of servers (servnets in Free Haven terminology) that preserves stored files for reading, but not for updating. The servers trade their available storage space, and therefore are able to store files on the other servers in the network. A client that wants to insert a file should apply erasure coding first, and then choose one of the servers. After receiving the file, the server can store the file locally, or move it somewhere else. Finding a file is not particularly scalable. Its ID must be known, and if the contacted server does not have it, it broadcasts the request to all servers it knows until the file is found.

Backup systems are a special category of archival systems. Usually, backups are stored on a dedicated server that provides massive storage capacity. In order to avoid such an expensive node in the system, Batten et al. [BBST01] came up with the idea of managing backups on top of the Chord network. Files for backup are encrypted, divided into chunks and stored within the network. In order to increase data availability, file chunks are replicated and stored on different peers. However, the authors do not impose any requirements on the replication factor. It is left up to the user.

An interesting backup approach was presented by the authors of Pastiche [CMN02], implemented on top of Pastry. Namely, Cox et al. observe a set of similar machines, whose files should be preserved. Since a big percentage of files is already the same on all machines, they will not be stored again in the network. All files are indexed and chunked. Before storing a file

within the network, a peer checks if some chunks are already on some of the other machines, and given a positive answer those chunks are skipped. The chunks are immutable. If a file is changed, modified chunks will be stored again. The proposed approach reduces storage overhead significantly, if the machines have similar content. On the other hand, Pastiche expects that the machines are available most of the time. Otherwise, recovery is not possible.

CFS (the Cooperative File System) [DKK+01] is a P2P storage designed by Dabek et al. Files are split into a number of blocks, which are then distributed over a set of highly available CFS servers. The storage is built on the top of DHash that is based on Chord for locating the servers responsible for a block. CFS does not use erasure coding – files are chunked simply into n blocks. In order to reconstruct a file again, all n blocks are needed. To ensure that a block can be found with a high probability, it is replicated on k CFS servers. Unfortunately, the authors do not provide any insight into how a proper value for k is determined. Since the availability of CFS servers is pretty high, it is assumed that the number of replicas is kept constant over time. Departure of a CFS server can be detected and all blocks managed there can be recreated elsewhere. The proposed storage can be considered as read-only, since only the file creator is able to modify it later.

Rowstron and Druschel applied very similar ideas in building PAST [RD01b] – a write-once/read-many storage. Instead of Chord, they selected Pastry as the underlying DHT overlay. Unlike in CFS, files are not chunked in blocks – the PAST authors wanted to reduce the time needed for locating and retrieving them. To keep data highly available, files are replicated at k closest peers in the leafset of a destination peer. The authors do not provide a deeper analysis about the sufficient replication factor. It is assumed that the peer availability is pretty high and stable, above 50%. All files' copies share the same ID. Peers in the leafset periodically exchange keep-alive messages. Thus, going offline can be detected, and if a file replica stays offline for a longer time, it is recreated.

As one can see, choosing the right replication policy is usually hard. Thus, Brodsky et al. [BBP+02] propose the Mammoth file system, where administrators or users decide about the preferred replication policy for given files or group of them. Updates are possible, but only eventual consistency is guaranteed.

Weak Consistency Models

Unlike strict consistency, weak consistency allows updates even if some replicas are not available at that moment. This is seen as a solution for applications that need to perform updates even when some machines are down, or unreachable due to network partitioning. If they perform updates frequently, they could experience better performance, because an update confirmation can be sent before all replicas are modified.

A disadvantage of weak consistency is the lack of any guarantees about the length of the inconsistency period. Some of the available replicas might be obsolete, but application that reads them might not be aware of this. To protect against inconsistent access, only up-to-data replicas should be accessible, but this would decrease the availability of data.

As mentioned in Section 3.2, the availability of peers in an Internet-wide P2P system is much lower than the one assumed in the distributed data management. Therefore, accessing only the consistent replicas leads to a significantly reduced data availability, which is not acceptable for many scenarios. In order to provide high data consistency and availability under a weak consistency model, many P2P systems assume that at least some peers are highly available.

If accessing up-to-date data is not crucial for proper application behavior, weak consistency is a better choice. Petersen et al. [PST+97] present the anti-entropy protocol for update propagation between weakly consistent replicas. The protocol is implemented as a part of Bayou storage. It is based on pair-wise communication, the propagation of write operations, and a set of ordering and closure constraints. The protocol is very flexible, since it has no requirements regarding the network environment and uses minimum communication bandwidth. The application can update any replica, and the update is propagated further until eventually all replicas are updated. The goal of anti-entropy is for two replicas to bring each other up-to-date. How replicas find each other (i.e. replica update policy) is not part of the protocol, and it can be done in many different ways in practice. Peers holding the replicas keep ordered logs of all write operations that the replica has seen. Therefore, anti-entropy enables two peers to agree on the set of writes stored in their logs. Write propagation is constrained by the accept-order, i.e. a peer accepts only write operations that are newer than the last write operation in its log, accepted by some other peer.

The consistency schema presented in this thesis is also weak. Unlike the other approaches, our protocol delivers the requested consistency level immediately after the update. Eventually, all unreachable replicas become up-to-date, making data fully consistent.

As demonstrated by Terry et al. [TTP+95], the write logs contain enough information to detect when concurrent updates produce a conflict. Conflict resolution can be solved by applying a defined merge procedure. However, the procedure does not work in a generic case, because the semantics of data in conflict must be known. Alternatively, Bayou can mark the conflicting replica, and provide this information to an application hoping that it or the user can resolve the conflict. The disadvantage of this approach is that conflicts can only be detected quite late, even though the application running on a peer has been informed that the update has been successful.

The update mechanism presented in this thesis guards against conflicts during an update, and refuses to complete the request if a problem is detected. It does not provide an automatic conflict resolution. Therefore, the application is informed, and it should compensate the request somehow.

Datta et al. [DHA03] describe a weakly-consistent update mechanism for P-Grid [Abe01] – a DHT variant. It is based on so called rumor spreading, but is modified to achieve a lower communication overhead. At the same time, it should support systems with hundreds of replicas. The protocol tries to shorten the time needed for updating all replicas. Therefore, it consists of two phases. An initiator of update starts the push phase, and the update request is sent to a subset of peer that have the replica. The peers that receive the update request propagate it further to a subset of peers that have the replica as well. Peers coming online, or those that did not receive any update for a long time can enter into the pull phase. They find other replicas and try to

synchronize as in anti-entropy. The authors assume that the probability of having concurrent updates is low, and therefore the protocol does not have any support for conflict resolution.

The update mechanism presented in this thesis is able to address directly all available replicas, so the communication overhead is lower than by P-Grid. The update version of unreachable replicas will be stored on some peers in the network. Unlike in P-Grid, peer that were disconnected do not attempt to initiate synchronization of replicas. Instead, the other peers are able to notice that DHT topology has changed, and they will send all replicas that are not under their responsibility to the peers that are actually responsible.

Storages and File-systems

FarSite [ABC+02] aims to provide strict file-system semantics to a client. The file-system itself is physically distributed among a set of untrusted, but highly available machines. A distributed directory service, known to all machines, is used to locate content. Data availability is achieved by replication. Updates are possible and realized using Byzantine agreement. This is possible only when more than two thirds of replicas are online. To summarize, much of FarSite design expects that the environment has properties similar to a LAN. Hence, it is not really suitable for wide-area deployment, with slower and unreliable communication links, and lower peer availability.

RepStore [LJ04] is another P2P storage based on top of DHT, targeting enterprise LANs (i.e. it requests stable and high peer availability). It is self-tunable, i.e. it tries to achieve the best cost/performance tradeoff by devoting higher storage overhead to frequently updated files. Namely, such files are replicated, and the rest are encoded using erasure coding.

Oceanstore [KBC+00, RWE+01, Wel00] is a P2P file-storing system developed at the University of California, Berkeley, aiming to provide a wide-area storage system. It has a super-peer based architecture. For archival purposes, data are encoded first using Reed-Solomon codes (erasure coding), and then spread amongst super-peers using Tapestry [ZKJ01, ZHR+04] DHT overlay. For the operational usage, data are kept in the native form replicated across the network. Since it is assumed that the super-peers are highly available, the authors of OceanStore do not provide any analysis about the needed replication factor. The replication is mostly used for speeding up data access, i.e. placing replicas closer to peers that request them. The storage is self-repairable – if a servers crashes or goes offline, the rest of the super-peers are able to reconstruct all missing fragments. Updates are supported and are based on versioning of objects, and master/slave approach. An update request is sent first to the object's inner ring (primary replicas), which performs a Byzantine agreement protocol to achieve fault-tolerance and consistency. When the inner ring commits the update, it multicasts the result of the update down to the slave replicas.

Similar to our approach, Bhagwan et al. [BTC+04] recognized a need to specify the requested data availability as a part of configuring the storage. TotalRecall automatically measures and estimates the availability of its constituent host components, predicts their future availability based on past behavior, calculates the appropriate redundancy mechanisms and repair policies, and delivers user-specified availability while maximizing efficiency. The system measures short-term (transient errors) and long-term (non-transient errors) availability of participating peers. How-

ever, the authors do not provide details of how the measurements are performed, and what the obtained precision is. TotalRecall can manage mutable data. An update produces a new data version, stored separately in the system. Periodically some sort of garbage collection is performed and the obsolete data versions are removed. The system distinguishes master and slave replica copies. An update is first performed on a master responsible for an object. Afterwards, the master updates all slaves. Our approach is simpler, we do not distinguish between master and slaves, so there is no need to select a new master when the old one goes offline. Even during an update, peers could go offline, and if there is no conflict, the update will be successful.

Instead of estimating the peer availability and choosing the appropriate replication policy, Pangaea wide-area file system [SKKM02] replicates data aggressively, i.e. every time a peer requests an access to a file that cannot be found locally, it is copied there. In an extreme case when all files are accessed at least once by all peers and the local storages are large enough, the storage will be fully replicated. Pangaea allows updates on the storage files. It is enough to update any of the replicas (usually the local one), and the changes will be propagated in the background. Potential conflicts are detected and resolved after they happen. Thus, the storage supports only eventual consistency, not being able to give any probabilistic guarantees on consistency of data access after update. In case of concurrent updates, the system can apply two strategies: last writer wins, or manual conflict resolution.

Ivy [MMGC02] is a mutable peer-to-peer file system that enables writes by maintaining a log of changes. The logs are managed by using a DHash DHT implementation. Reading up-to-date file version requires the consultation of all logs, and that is not very efficient. To speed up performance, peers can make snapshots of frequently used files to avoid log scanning. Surprisingly, data are not replicated, nor recoded using erasure coding. Hence, keeping high data availability requires that peers are constantly online. If some them go offline, many files cannot be reconstructed, and therefore become unavailable. Conflict during updates might happen, and an additional tool has been provided that can be run manually in order to resolve conflict.

Yu and Vahdat present Om [YV05], a peer-to-peer file system that achieves high data availability through automatic replica regeneration while still preserving consistency guarantees. Every managed object is described by a configuration that defines locations of primary and secondary replicas. Data access is performed by using a read-one/write-all quorum, i.e. implicitly high peer online probability is assumed. All writes are first performed on the primary replica. Afterwards, the update is propagated to secondary replicas using a two-phase commit protocol. If some of the secondary replicas are not available, regeneration is triggered, i.e. new replicas will be created and the configuration will be changed. In order to allow for inconsistencies, all available replicas must agree about the same new configuration, and for this purpose a quorum based on the witness model is used. The quorum intersection is not always guaranteed, but it is extremely likely.

Detecting that some replicas are not available requires that all live replicas in the configuration periodically probe other replicas. Since the number of configurations is equal to the number of managed objects, the traffic related to probing increases significantly as the number of managed objects increases. Om creates new replicas every time a replica disappears. The authors

did not provide enough details about what happens to data on a peer that rejoins. If data are preserved, the number of replicas in the configuration increases. If the average peer availability increases during some periods, the size of the configuration will be much higher than really needed for maintaining high data availability. Also, the authors do not provide any details about the configuration size needed to guarantee high data availability.

The approach in this thesis does not require the existence of master and slave replicas. An update addresses all available replicas in a predefined order guarding against concurrent updates as well. In order to maintain the requested level of availability and consistency, peers need to measure the actual peer availability. However, the traffic generated is much lower than in Om. The number of messages is fixed per peer and independent of the number of locally stored replicas. If the measured peer availability is not sufficient to achieve the requested goals, the number of replicas will be increased. On the other hand, when data availability is above the defined level, the number of replicas will be reduced to the minimum needed, depending on the actual peer availability.

Continuous Consistency Model

Yu and Vahdat [YV02] recognize that there is a continuum between a strong and weak consistency model that is semantically meaningful for a broad range of applications. In order to bound the maximum rate of inconsistent access, the authors developed a set of metrics: numerical error, order error, and staleness. The numerical error limits the total weight of writes that can be applied across all replicas before being propagated to a given replica. The order error limits the number of tentative writes that can be outstanding for any replica, and staleness places a real-time bound on the delay of write propagation among replicas. The proposed continuous consistency model is able to describe both strong and weak consistency models. Namely, they represent two extremes: if all parameters are set to zero, we get the strong consistency model, while the weak consistency model is defined with all parameters being infinite.

The continuous consistency model was implemented by application middleware called TACT [Yu00] that mediates application read/write access to a fully replicated data store. The maximal allowed inconsistency of a managed replica is defined by an instance of the proposed metric. When an update comes, the peer checks the potential inconsistency level if the update is applied. If it is below the defined maximum, the update is accepted and propagated, otherwise it is refused. Accepted updates are propagated to other replicas by using the anti-entropy approach. The drawback of the proposed approach is that consistency levels allowed must be specified manually by application, and for every data type separately.

The approach presented in this thesis limits data inconsistency as seen by application by specifying the consistency level of updated data, until all obsolete data are eventually synchronized. Unlike TACT, the application is not involved in the process of specifying the guarantees. Our storage provides the same level of inconsistency to all managed data, independent of the application logic.

3.4.3 Placement

Although data placement is out of the scope of this thesis, the following section proves that it is still an important issue in P2P data management. If replicas are stored on peers close to those that need them, or on peers with good network bandwidth, an application could experience a significant gain in performance.

The approach presented in this thesis relies solely on the data placement implemented by the underlying DHT implementation. Hence, the one with the most suitable data placement can be used.

Existing solutions usually assume that P2P storages manage immutable data. Thus, access performances can be increased through caching. Storages such as PAST [RD01b] and CFS [DKK+01] already cache data on all nodes on the path from the query destination back to the query source. However, Gopalakrishnan et al. [GSBK04] state that such a schema has a high overhead even under moderate load. They propose a solution that relies on server load measurements to decide whether and where new replicas should be stored. The routing process is augmented, so that a request can be diverted quickly towards new replicas.

Ramasubramanian and Sirer [RS04] designed Beehive – an extension to a DHT overlay that has $O(1)$ routing complexity, i.e. it finds all data in one hop. As we have mentioned in Section 3.2, the number of hops to the final destination in DHT usually follows $O(\log N)$ complexity. Beehive makes the routing paths shorter by using proactive replication, i.e. propagating replicas within the network. To limit the storage overhead, the maximum number of replicas for each object is limited as well.

3.5 Summary

This chapter has given an overview of the state of the art in handling availability and consistency of data in distributed and decentralized systems. Significant research was conducted in the past decades, addressing and solving many important issues. Although some ideas and results achieved can be applied to decentralized/P2P data management, many of them cannot, because they are built on assumptions that the system/environment has some properties which are usually not present in decentralized/P2P settings. Therefore, achieving high availability and consistency of data in such an environment requires the development of new solutions.

Since both the classical distributed and the decentralized/P2P system are classes of distributed systems in general, it is necessary to understand better the relationship between them. Section 3.1 has categorized all distributed systems into several classes according to the following criteria: availability of components, communication patterns among components, quality of communication links.

Section 3.2 presented in detail further different P2P architectures, and points out their pros and cons. Since they have the most advantages, the focus of the Section was on structured overlays – in particular DHTs.

Section 3.3 has presented approaches for dealing with the availability and consistency of data in classical distributed data management. Both replication and erasure coding were introduced, pointing out the differences. Replication protocols that ensure the implementation of different consistency models have been covered: strong (ROWA and its derivates, quorums), weak (master-slave, primary-secondary). Based on how they deal with network partitioning, we have presented optimistic/pessimistic classification as well. Finally, although not a topic of this thesis, the section has given an insight into data placement techniques that are very important for achieving good access performances.

Since the properties of a decentralized/P2P system are quite similar to those of classical distributed systems, in many cases decentralized data management adopts existing techniques. As presented in detail in Section 3.4, data availability is achieved by replication or erasure coding. However, most of the time, getting high data availability is done under the assumption that peer availability is fairly high, stable, and known in advance. In such a case, a decentralized/P2P system does not differ much from a classical distributed system. Unfortunately, as popular P2P systems show, these assumptions cannot always hold, and the methods applied will not deliver the requested results. Maintaining the consistency of data is even harder. As can be seen, the strict consistency model cannot be guaranteed, and the following options are possible: no consistency at all, weak and continuous consistency models. The existing weak consistency models provides guarantees that all replicas will be eventually updated, but they cannot provide any consistency guarantees on accessed data until this does happen.

The approach presented in this thesis manages high data availability in a self-adaptable way and provides probabilistic guarantees of consistency of data that are accessed after update. If the requested data availability is not present in the DHT, the proposed replication protocol will increase the number of replicas until availability is successfully reached. The protocol is based on fewer assumptions than the existing solutions. It can be deployed in a DHT with an arbitrarily low peer availability. No knowledge about the underlying DHT is needed in advance. Also, the requested guarantees can be recovered or even maintained after or during churns.

Chapter 4

Approach

THIS chapter describes in detail the replication protocol that solved the problem defined in Section 1.2. As presented in Section 3.3, replication protocols need to have access to a replica directory. State-of-the-art solutions propose placing such a directory on a well-known, reliable machine. In general, DHTs do not have such a reliable and always available peer that could take on this role. Hence, designing a decentralized replica directory is seen as the only option. Originally proposed in [KWRF05] by Knežević et al., Section 4.1 presents it in detail.

Section 4.2 subsequently defines a transparent replication mechanism suitable for immutable data. The analysis of the protocol introduced clearly shows how the number of replicas depends on the availability of requested data and the actual peer availability. The first version of the protocol appeared in [KWRF05]. The version presented in this Section has been published before in [KWR06b].

In order to support management of mutable data, Section 4.3 extends the protocol to allow updates. The supported consistency model is the weak one. Unlike the state-of-the-art solutions, we provide requested consistency guarantees in a period in which all replicas are not yet updated. Again, the section analyzes dependency among the number of replicas, requested consistency and availability level, and the actual peer availability. The first ideas about the proposed consistency model were introduced in [KWR05], reaching their current status in [KWR07].

The performed analysis shows that self-adaptation can be achieved only if peers are able to measure the actual peer availability with a good precision. Section 4.4 presents in detail how to do that in a fully decentralized way and with moderate costs. The measurement technique was originally published in [KWR06b], but it turned out that the results obtained were precise only in the case of immutable data. This issue was soon resolved and the final version was published in [KWR07].

Finally, we are able to adjust the number of replicas both for immutable and mutable data. As first introduced in [KWR06b], Section 4.5 describes how to keep the number of replicas to the minimum needed to guarantee the requested data availability and consistency.

At the end of this Chapter, Section 4.6 discusses the protocol overhead, i.e. introduced storage and communication costs. The estimation is given both for immutable and mutable data, comparing them as well.

4.1 Decentralized Replica Directory

Every replication protocol needs to have access to a directory that maintains information about replicas and their location. Namely, by querying the directory with an object ID, it should return a list of all locations where its replicas can be found. Afterwards, it is up to the requester to decide what to do with the information obtained. For example, reading would require accessing any of the replicas, whereas writing should update all/some of them by using a defined mechanism. When a peer inserts an object into the system and replicates it a number of times, it must create a new directory entry containing all replica locations. Finally, if the number of replicas of an object changes in the system, the proper directory entry must be updated.

4.1.1 Explicit Replica Location Management

The directory could be easily realized by using a hash table data structure. The hash tables manage (*Key*, *Value*) pairs that can be used for tracking all locations where a replica is stored. Namely, the *Key* could correspond to an object ID, and the *Value* could contain the addresses of peers that own a replica.

Similarly, because a DHT provides the same functions as a hash table, the decentralized replica directory could be implemented on the top of it. Since routing is deterministic and key-based, if a replica is stored in the system under a key, its value is interpreted as the replica location. In order to avoid conflicts, all replicas of all managed objects in a DHT must be inserted under different, unique keys. After inserting all replicas, information about their locations could be stored in the DHT under *Key* that takes the value of the object ID. Finding a previously stored object could be realized by finding the *Value* stored under the object ID, reading the replica locations, and issuing another request, this time using some of the replica *Keys* found.

The idea described has two significant drawbacks. Since directory entries are spread among peers, when a peer goes offline, all objects associated with locally managed directory entries immediately become inaccessible, even if some of the replicas are still available on other online peers. Such scenarios can happen often, since an object ID differs from all replica IDs, meaning that it is quite usual for a directory entry and replicas to be stored on different peers in the DHT. On the other hand, when a directory entry is available, finding an object requires issuing at least two requests; the first one returns the list of all replica keys, and the second one returns the replica itself.

4.1.2 Implicit Replica Location Management

The issues described can be solved by managing directory entries implicitly within a DHT. Namely, every peer should be in a position to locally generate the list of all replica keys for an object. If this is fulfilled, the explicit directory entries are no longer needed in the DHT. An object would be available as long as at least one of its replicas is available too.

To summarize, every managed replica is associated with a key used for storing it in the DHT. The first replica key is equal to the object *Key* and is obtained from the application or generated using a random number generator. All other replica keys are correlated with the first one, i.e. they are derived from it by using the following function:

$$replicaKey(Key, ron) = \begin{cases} Key & : \quad ron = 1 \\ hash(Key + ron) & : \quad ron \geq 2 \end{cases} \tag{4.1}$$

where *ron* is a replica ordinary number, *hash* is a good hash function with a low collision probability, such as MD5 [Riv92] or SHA-1 [EJ01]. Since hash values are produced from an arbitrary byte array, *Key* and *ron* are observed as two byte arrays concatenated via the $+$ operator. If the keys are *b*-bits long, and assuming that *Key* of two objects are different, the probability of having the same value on any of the replica keys is 2^{-b}. Since MD5 and SHA-1 hash functions generate 120 and 160-bit long keys, this probability is equal to 2^{-120} and 2^{-160} respectively.

The proposed key generation schema ensures that all replicas are stored under different keys in the DHT. Further, a consequence of using a good hash function is that two consecutive replica keys are usually very distant from each other in the key space. Changing a bit on the input produces a new hash value that is very different to the previous one. Since every peer in a DHT is responsible only for a part of the whole key space, it means that replicas of an object should be managed by different peers in a fairly large network. Our empirical evaluation has proven that this assumption is valid for the large majority of managed objects. A drawback of the proposed schema is an inability to influence the replica placement. The replica location solely depends on the obtained replica key and the used DHT implementation. Thus, some replicas might be faraway from peers that could need them often.

In order to find the replica of an object, an application needs to provide the DHT with only one object key; the remaining replica keys will be computed locally and online replicas could be accessed. As we are going to see later, creating more replicas of an object whose replica is managed by a peer requires that the peer has access to the object's *Key*. On other hand, unneeded replicas are removed from the system based on their ordinary number. This information should be attached to the stored replica. Therefore, a replica will be wrapped in a tuple $(Key, ron, Value)$ that is inserted later into the DHT.

4.2 High Availability of Immutable Data

The following Section describes how to guarantee high availability of immutable data. The main challenges are:

Algorithm 1 $store_{immutable}(Key, Value)$

Require: $R > 0$

1: **for** $i = 1$ to R **do**
2: $rk \leftarrow replicaKey(Key, i)$
3: $t \leftarrow (Key, i, Value)$
4: $store(rk, t)$
5: **end for**
6: return SUCCESS

- How to make replication fully transparent to applications
- How to determine the number of replicas needed to manage the requested data availability

The protocol must provide the same data availability for all managed objects (see Section 1.2), and therefore the number of replicas R must be the same for all objects. Joining the community for the first time, a peer assumes an initial value for R, or obtains it from its neighbors.

4.2.1 Transparency

Managing high availability of data transparently requires that the DHT equipped with our replication protocol exposes the same functions as before. Storing ($store_{immutable}(Key, Value)$) a value $Value$ under the given key Key requires that the value is transparently wrapped in R tuples $(Key, ron, Value)$, where $ron = 1 \ldots R$. These tuples are then inserted into the DHT under R different keys obtained using Formula 4.1. From an application point of view, $Value$ is inserted into the DHT under Key.

In order to access a value ($lookup_{immutable}(Key)$) stored previously under key Key, it is sufficient to find any available replica. Using the key provided as a basis, the replication protocol generates R replica keys using Formula 4.1, and tries to find at least one of the previously stored tuples. If found, $Value$ is extracted from the tuple, and returned to the application.

Algorithm 1 and 2 present the pseudo-code for the transparent replication mechanism described. The number of replicas R should be known, and must be sufficient for the requested data availability.

Although the names of the defined functions are marked with *immutable* in the index, from an application point of view they have the same signature and functionalities as those introduced at the beginning of Section 1.1. Moreover, the functions provided keep data highly available and fully transparent. Therefore, using them does not require any changes within the application code. It can make the application code even simpler, if it were dealing with the availability of data as well. Such parts of the code will not be needed anymore, if the proposed replication mechanism is in use.

Algorithm 2 $lookup_{immutable}(Key)$

Require: $R > 0$

```
1: for i = 1 to R do
2:     rk ← replicaKey(Key, i)
3:     t ← lookup(rk)
4:     if t ≠ null then
5:         return t.Value
6:     end if
7: end for
8: return null
```

4.2.2 Number of Replicas

The following analysis models the availability of data in a DHT, assuming that they are replicated R times. Let us denote with p_r the probability of a replica being online. Since both replica keys and peer IDs are generated randomly, in a fairly large DHT (see Section 4.1) we can assume that after a $store_{immutable}$, every replica will be placed on a different peer, i.e. $p_r = p$, where p is the actual peer availability.

Let us define now with Y a random variable that represents the number of replicas being online for a given object, and $P(Y = y)$ is the probability that y replicas are online. Given an assumption that peers behave independently and identically, the probability that y replicas are online at any point in time can be given as:

$$P(Y = y) = \binom{R}{y} p^y (1-p)^{R-y} \tag{4.2}$$

The average number of replicas R_{avg} available at any point in time is the expectation of variable Y:

$$\begin{aligned}
R_{avg} &= \mathsf{E}(Y) = \sum_{y=0}^{R} yP(Y=y) \\
&= \sum_{y=0}^{R} y \binom{R}{y} p^y (1-p)^{R-y} \\
&= \sum_{y=1}^{R} y \frac{R!}{y!(R-y)!} p^y (1-p)^{R-y} \\
&= \sum_{y=1}^{R} y \frac{R}{y} \frac{(R-1)!}{(y-1)!(R-y)!} p p^{y-1} (1-p)^{R-y}
\end{aligned}$$

$$
\begin{aligned}
&= pR\sum_{y=1}^{R}\frac{(R-1)!}{(y-1)!(R-y)!}p^{y-1}(1-p)^{R-y} \text{ , rename } m=R-1 \text{ and } s=y-1\\
&= pR\sum_{s=0}^{m}\frac{m!}{s!(m-s)!}p^{s}(1-p)^{m-s}\\
&= pR\sum_{s=0}^{m}\left(\begin{array}{c} m \\ s \end{array}\right)p^{s}(1-p)^{m-s}\\
&\quad \text{the sum is a sum over complete mass probability function, thus equal to 1}\\
&= pR \qquad (4.3)
\end{aligned}
$$

As defined by $lookup_{immutable}$, an object is available if at least one of its replicas is online. Therefore, the data availability a can be expressed as:

$$
\begin{aligned}
a &= P(Y \geq 1)\\
&= 1-P(Y=0)\\
&= 1-(1-p)^{R} \qquad (4.4)
\end{aligned}
$$

The number of replicas R needed is

$$R = \left\lceil \frac{\log(1-a)}{\log(1-p)} \right\rceil \qquad (4.5)$$

The Formula clearly says that the requested data availability a can be delivered in a DHT, where peers are available with the probability p only if data are replicated at least R times. As presented in Section 3.4, peer availability can and does vary in a reality. Thus, guaranteeing the requested data availability over time is possible only if the actual peer availability can be measured.

4.3 High Availability of Mutable Data

Having the ability to modify stored objects is essential for a wide range of applications. Without updates, the life of a replica is very simple; the replica could be offline or online, but its value is always correct. By replicating mutable data, consistency becomes the main issue. There are potentially two sources of inconsistency:

- Replicas are not reachable due to network partitioning or being offline
- Uncoordinated concurrent updates

As discussed in Section 3.4.2, strict consistency models are usually not suitable for P2P data management, because they require highly available peers or a super-peer network to deliver acceptable performance. Otherwise, enforcing strict consistency in a low-availability network would make updates impossible most of the time – not usually acceptable for many applications.

Requesting that our protocol can be applied on a DHT with an arbitrary peer availability makes weak consistency the only option. Although all replicas will eventually be up-to-date, the major disadvantage of the existing weak consistency models is that they do not provide any guarantees on the consistency of updated objects before all replicas become synchronized. Application gets either a correct or obsolete object's value, but the models applied are not able to provide any probabilistic guarantees on finding the correct value.

Unlike the existing approaches, the goal of the consistency model presented in this thesis is to provide probabilistic guarantees of the consistency of modified data until all replicas are not eventually synchronized. The model guarantees that the returned value will be correct at least with the given probability. At the same time, the protocol guards against concurrent updates.

Similar to ROWA-A, all available replicas will be updated. In addition to that, the new version of missing replicas is inserted into the DHT during update. The unavailable replicas will synchronize eventually using the push method (correct replicas inform obsolete replicas of the change). By looking up stored data, our protocol does not read just any of the available replicas – it tries to find the most up-to-date one among all those available. Therefore, apart from information already stored in managed tuples, the *version* of a replica and its *age* must be there as well. To summarize, if mutable data are to be managed in DHT, a replica will be wrapped in a tuple $(Key, ron, version, age, Value)$.

4.3.1 Transparency

The protocol defined in Section 4.2 is modified slightly while data are kept highly available and consistent in a transparent way. When a value is about to be inserted in a DHT ($store_{mutable}(Key, Value)$), it is wrapped in R tuples, appropriate keys are generated (Formula 4.1), and the version number is assigned to 0. Any age value set up by the initiator of the request is ignored on the receiver side. Upon receiving it, the age becomes the value of the current local time, which will be used later on as basis for producing correct age information. With every update, the version number is incremented by 1. All online replicas are updated, and the new version of all offline replicas inserted in the DHT. If a replica is offline, the peer that should be responsible for it at the moment when the update takes place becomes the new version.

Every time a $store_{mutable}$ is invoked, it must be detected what the next replica version number should be. If a new object is inserted, the version number should be zero, otherwise it must be incremented. Therefore, a log of (*Key*, *Version*) pairs of successful lookups must be kept locally.

Concurrent Updates

Concurrent updates control defines that replicas are updated in a predefined order, i.e. a sequence: first update the 1^{st} replica, then the 2^{nd} replica until the R^{th} replica. If the update of any replica fails, the update stops and the rest of the replicas are not touched. The update fails if a peer that receives the new replica already has a replica with a higher or the same version number. Implicitly, updating an object requires reading it before, which is a common pattern in the majority of applications: data are read first, and based on their value and implemented logic, the new value is generated. If data are not read first, the protocol does not find the corresponding (*Key*, *Version*) pair in the log, and believes that the object is about to be inserted. However, this fails, because replicas under the same keys are already in DHT. Their version number is at least zero, which is higher than or equal to the version of replicas that are about to be inserted.

If more peers want to update an object concurrently, they all must follow the described update procedure. They send update requests in the defined order, and continue only if a replica is successfully updated. The update requests on a destination peer are also processed sequentially. Only the first request gets through, the rest of them fail. After the successful update, the replica with the same version exists already in the storage. The originators of these requests are informed about the failure and they must be compensated. A feasible compensation could be to inform the user about the problem and give up, or re-read the actual value, and try to update it again.

When a peer wants to access an object, it is not sufficient just to return any available replica. Instead, we should return the replica with the highest version number to ensure that the application gets the most up-to-date version. However, if two or more replicas with the same version (e.g. as a result of network partitioning), but with different values are found, the youngest replica (i.e. with the lowest age value) will be returned. A receiver of a lookup request is able to produce a correct age value by subtracting the actual local time from the time when the replica was stored in its local storage.

Algorithms 3, 4, 5, and 6 present the pseudo-code of the described replication mechanism. As in Section 4.2, high availability and consistency of data are managed in a fully transparent way, i.e. without a need to modify the application that is built around DHT functionalities described in Section 1.1.

Note that the defined $lookup_{mutable}$ does not ensure that the obtained replica is indeed the correct one. Only if a replica with the latest version number is available, does the application get the correct value, i.e. up-to-date objects. The analysis presented in Section 4.3.2 shows how to ensure that the returned value is up-to-date with an arbitrarily high probability.

The proposed replication mechanism does not require that peers' clocks are synchronized, which would be almost impossible to achieve in the reality anyway. Most of the time, the freshest available replica is detected just by its version number, which is fully independent from time. Only due to network partitioning, found replicas might have the same version number but different content. In such a case, the freshest replica is the one with the lowest age. Its value is produced only by using the peer's local clock that must not be synchronized with the rest of

Algorithm 3 $store_{mutable}(Key, Value)$ (initiator side)

Require: $R > 0$ {originator side}

```
1: v ← getVersionFromLog(Key)
2: if v ≠ null then
3:     v ← v + 1
4: else
5:     v ← 0
6: end if
7: for i = 1 to R do
8:     rk ← replicaKey(Key, i)
9:     t ← (Key, i, v, 0, Value)
10:    status ← store(rk, t)
11:    if status = conflict then
12:        return CONFLICT
13:    end if
14: end for
15: return SUCCESS
```

Algorithm 4 $store_{mutable}(Key, Value)$ (receiver side)

```
1: Value.age ← currentTime
2: if Key exist then
3:     t_stored ← getFromStorage(Key)
4:     if t_stored.Version ≥ Value.Version then
5:         return conflict
6:     else
7:         updateInStorage(Key, Value)
8:     end if
9: end if
10: putInStorage(Key, Value)
```

peers in DHT.

Receiving Missing Updates

When a peer rejoins the DHT, it does not change its ID, and as a corollary the peer is now responsible for a part of the keyspace that at least intersects with the previously managed part. Therefore, the peer keeps data in its storage and no explicit data synchronization with other peers is required. Replicas whose keys are no longer in the part of the keyspace managed by the rejoined peer can be removed and sent to peers that should manage them according to the current

Algorithm 5 $lookup_{mutable}(Key)$ (initiator side)

Require: $R > 0$

1: $version_{max} \leftarrow -1$, $age_{max} \leftarrow -1$, $value \leftarrow null$
2: **for** $i = 1$ to R **do**
3: $rk \leftarrow replicaKey(Key, i)$
4: $t \leftarrow lookup(rk)$
5: **if** $t \neq null$ and $t.Version > version_{max}$ or ($t.Version = version_{max}$ and $t.age < age_{max}$)) **then**
6: $value \leftarrow t.Value$, $version_{max} \leftarrow t.Version$, $age_{max} \leftarrow t.age$
7: **end if**
8: **end for**
9: put $(Key, version_{max})$ in lookup log
10: return $value$

Algorithm 6 $lookup_{mutable}(Key)$ (receiver side)

1: $t_{stored} \leftarrow getFromStorage(Key)$
2: **if** $t \neq null$ **then**
3: $t_{stored}.age \leftarrow currentTime - t_{stored}.age$
4: **end if**
5: return t_{stored}

DHT topology. If a peer already has an older replica and receives a newer one, it is practically informed about some missing updates.

4.3.2 Number of Replicas

All immutable objects managed by a DHT are always fully consistent. Hence, we were able to define in Section 4.2 that an object is available if any of its replicas are available. Based on this, we have computed the number of replicas that is necessary to deliver arbitrarily high data availability guarantees.

This is not enough if updates are allowed. Finding any replica is not sufficient, because it does not guarantee that the replica found is the latest one. Therefore, we define that an object is available if at least one of its latest correct replicas is available.

The life of an object begins with storing its R replicas in the DHT. With the first and every subsequent update, the protocol is able to address only the replicas that are online at that moment. Offline replicas are not reachable, but their new version is stored on peers that should keep them according to the current DHT topology. Finally, after an update, the DHT contains again R correct replicas of an object. Due to inserting the new version of offline replicas in the DHT, the total number of replicas R_t managed by the DHT for a given mutable object is not R anymore.

Offline replicas cannot be addressed during an update, and thus the new version will be placed on some peers in DHT under the same keys. Hence, all replicas of an object are still managed under R different keys.

Under an assumption that peers are independent and behave similarly, the number of replicas that an update addresses on average is pR_t (see Formula 4.3), where p is the average peer availability. After a successful update, the rest of $R_t(1-p)$ replicas become incorrect, and the total number of replicas is increased by $R_t(1-p)$ replicas. However, the total number of replicas is bounded by the DHT routing mechanism. Since replicas are managed under R different keys, then the maximum number of accessible replicas cannot be greater than R, i.e. $pR_t \leq R$.

Let us denote with p_r the probability of having a correct replica available. Since both keys and peer IDs are generated randomly, we can assume that after a $store_{mutable}(Key, Value)$, every replica will be placed on a different peer, i.e. $p_r = p$. Let us denote with Y a random variable that represents the number of freshest replicas being online for a given object, and $P(Y = y)$ is the probability that exactly y freshest replicas are online. Under an assumption that peers are independent, the probability can be given as:

$$P(Y = y) = \left(\begin{array}{c} R \\ y \end{array} \right) p^y (1-p)^{R-y} \tag{4.6}$$

The probability a that a DHT value is available and consistent is equal to the counter probability that none of the correct replicas are online:

$$a \quad = \quad 1 - P(Y = 0) \tag{4.7}$$

Therefore, the number of needed replicas R is

$$R = \left\lceil \frac{\log(1-a)}{\log(1-p)} \right\rceil \tag{4.8}$$

The derived Formula is identical to Formula 4.5, i.e. the number of replicas that should be updated and inserted does not change if data are immutable or not. Guaranteeing the requested data availability and consistency over time would require measuring the average peer availability p, and adjusting the number of replicas of locally stored objects accordingly.

4.4 Measuring Peer Availability

Measuring actual peer availability is usually based on probing peers and computing a ratio between the number of positive and total probes. Ideally, finding the current value with absolute precision would require probing all existing peers. This is not really feasible in practice due to several reasons:

- Every peer has just a partial view of the whole system, i.e. a peer does not know how many other peers are member of the community, and therefore a full check cannot be done.
- Probing all peers does not scale in large communities, i.e. it consumes a lot of time and generates high network traffic.

Additional restrictions present in this thesis is that probing cannot be done directly (i.e. pinging peers) because we do not know anything about the peer community, i.e. peer IDs, and/or their IP addresses. Our protocol uses the underlying DHT only via the provided *route*, *store*, and *lookup* functions.

According to the properties of the key generation mechanism (Formula 4.1), we can assume that every replica of an object is stored on a different peer within a fairly large network. Hence, peer availability is equal to replica availability, and therefore it can be measured indirectly by measuring replica availability.

Since every stored tuple contains object *Key* as well, all keys of all other replicas and their availability can be checked. In a DHT that manages immutable data every replica is stored under a different key. Therefore, probing a replica implicitly checks the availability of a peer that owns it. Enabling updates, this approach is no longer valid, because different versions of the same replicas are managed under the same key. If v different versions of a replica exist in a DHT, the probability of having a replica with the given key online is $1-(1-p)^v$. It does not correspond to the peer availability p any more, and therefore probing must take into account the returned replica version as well.

A peer cannot know if a replica managed locally is the freshest one. However, we are free to start with such an assumption. The peer randomly chooses a replica from its storage, and uses it to randomly generate a key of another replica of the same object. If the selected replica is found, their versions are compared, and one of the following actions is taken:

- Versions are equal $\Rightarrow$ the probe is considered valid and trustworthy, i.e. its outcome is taken into account.
- The version number found is greater then the local one $\Rightarrow$ it signals that the peer it has an obsolete version that should be removed from its storage. The probing is not valid, and is not taken into account.
- The replica version number found is lower than the local one $\Rightarrow$ the probe cannot be considered as valid, and the remote peer is informed about this. Afterwards, the remote peer should remove the obsolete replica from its storage.

Probing with Requested Confidence

We cannot achieve absolute precision, and thus we are ready to accept some degree of error. Thanks to the confidence interval theory [BL95], it is possible to find out what the minimal

number of replicas that has to be checked is, so that the computed replica online availability $\hat{p}_r$ is accurate (close to the actual replica availability p_r) with some degree of confidence.

We make n valid probes and compute average replica availability $\hat{p}_r$ from:

$$\hat{p}_r = \frac{1}{n}(X_1 + \cdots + X_n) \tag{4.9}$$

where $X_i = 0$ when a replica is offline, or $X_i = 1$ for being online, and $1 \leq i \leq n$.

The expectation value and variance are then:

$$\mathrm{E}(\hat{p}_r) = p_r \qquad \text{and} \qquad \mathrm{Var}(\hat{p}_r) = \frac{p_r(1-p_r)}{n} \leq \frac{1}{4n} \tag{4.10}$$

where the inequality follows from the fact that $p_r(1-p_r)$ is maximized by $p_r = \frac{1}{2}$.

If the DHT size is large, we can approximate the distribution of $\hat{p}_r(X)$ by normal distribution

$$N\Big(p_r, \frac{p_r(1-p_r)}{n}\Big) \tag{4.11}$$

that has the same expectation value and variance. As known, any statistic $\bar{X}$ that can be described by a normal distribution of the following form

$$\bar{X} \sim N(\mu, \frac{\sigma}{n}) \tag{4.12}$$

can be further normalized to

$$\frac{\sqrt{n}(\bar{X} - \mu)}{\sigma} \sim N(0,1) \tag{4.13}$$

Using the above and since $\mu = p_r$ and $\sigma = \frac{p_r(1-p_r)}{n}$, we have

$$\frac{(\hat{p}_r - p_r)}{\sqrt{\frac{p_r(1-p_r)}{n}}} \sim N(0,1) \tag{4.14}$$

For a medium and small-size network, variance of $\hat{p}_r$ depends on the network size N

$$\mathrm{Var}(\hat{p}_r) = (\frac{N-n}{N-1})\frac{p_r(1-p_r)}{n} \tag{4.15}$$

and the following approximation is then valid

$$\frac{(\hat{p}_r - p_r)}{\sqrt{\frac{N-n}{N-1}\frac{p_r(1-p_r)}{n}}} \sim N(0,1) \tag{4.16}$$

Finding the probability that a value from the distribution $N(0,1)$ belongs to an interval $[a,b]$ with a probability C can be expressed as:

$$\mathrm{P}(a \leq N(0,1) \leq b) = C \tag{4.17}$$

If a statistic is approximated with the normal distribution as in Formula 4.13, the confidence of being within the given interval is then:

$$\mathrm{P}(a \leq \frac{\sqrt{n}(\bar{X} - \mu)}{\sigma} \leq b) = C \tag{4.18}$$

which can be rewritten as

$$\mathrm{P}(\bar{X} - \frac{b\sigma}{\sqrt{n}} \leq \mu \leq \bar{X} - \frac{a\sigma}{\sqrt{n}}) = C \tag{4.19}$$

Usually the interval is symmetric ($b = -a$), and to determine when a value is within the interval with the given probability C, $\Phi(b) \geq C$, where Φ is the distribution function of $N(0,1)$. For example, if $C = 95\%$ then $b = 1.96$ satisfies the previous inequation. Going back to the original statistic $\bar{X}$, we can define the appropriate interval there with the same confidence of 95%:

$$\left[\bar{X} - \frac{1.96\sigma}{\sqrt{n}}, \bar{X} + \frac{1.96\sigma}{\sqrt{n}}\right] \tag{4.20}$$

Based on the approximations presented, we are able to calculate the number of probes n we need to make in order that the computed replica availability p_r falls into the interval $(\hat{p}_r - \delta, \hat{p}_r + \delta)$ with a given probability C.

$$\mathrm{P}(\hat{p}_r - \delta \leq p_r \leq \hat{p}_r + \delta) \geq C \tag{4.21}$$

For variance and expectation as in Formula 4.10 (the DHT size is large), and applying the previously described approach, this can be rewritten as:

$$\begin{aligned}
\mathrm{P}(\hat{p}_r - \delta \leq p_r \leq \hat{p}_r + \delta) &= \mathrm{P}\left(-\frac{\delta}{\sqrt{\frac{p_r(1-p_r)}{n}}} \leq \frac{\hat{p}_r - p_r}{\sqrt{\frac{p_r(1-p_r)}{n}}} \leq \frac{\delta}{\sqrt{\frac{p_r(1-p_r)}{n}}}\right) && (4.22)\\
&= \mathrm{P}\left(-\sqrt{\frac{\delta^2 n}{p_r(1-p_r)}} \leq \frac{\hat{p}_r - p_r}{\sqrt{\frac{p_r(1-p_r)}{n}}} \leq \sqrt{\frac{\delta^2 n}{p_r(1-p_r)}}\right) && (4.23)\\
&\quad \text{approximating with Normal distribution } N(0,1)\\
&\approx \Phi\left(\sqrt{\frac{\delta^2 n}{p_r(1-p_r)}}\right) - \Phi\left(-\sqrt{\frac{\delta^2 n}{p_r(1-p_r)}}\right) && (4.24)\\
&\geq \Phi(\delta\sqrt{4n}) - \Phi(-\delta\sqrt{4n}) && (4.25)
\end{aligned}$$

The last step is valid because $0 \leq p_r \leq 1$, and therefore $p_r(1-p_r)$ has minimum in $\frac{1}{4}$.

Unfortunately, direct usage of confidence interval theory is not so promising for practical deployment. Namely, by requesting a small absolute error ($\delta = 0.03$), and a high probability of having the measurement within the given interval ($C = 95\%$), we have $0.03\sqrt{4n} \geq 1.96$ (since $\Phi(1.96) \geq 0.95$). It turns out that $n \geq 1068$, i.e. a peer should make at least 1068 random replica probes, generating high communication costs. Even if the network is not so large (e.g. 400 peers), the number of probes n drops only to 290.

Therefore, we need to slightly modify our probing strategy to achieve the requested precision with much lower traffic. A lower number of probes produces the results with higher variance, i.e. error. To get a good precision again, the final value should be produced by averaging more individual probing results. Therefore, along with the result of probing (replica available or not), the peer gets all the measurement values known by the probed peer when the probing request has ended. The result of the probing performed will be averaged with the received values, reaching the good precision again.

Getting the measurements done by other peers requires the introduction of two additional DHT message types. A peer that probes a replica sends a *PeerAvailabilityRequest* message with the replica key, and receives the answer via *PeerAvailabilityResponse* messages.

Computing the average peer online availability generates n *PeerAvailabilityRequest* and n *PeerAvailabilityResponse* messages, but a peer gets back measurements that have been able to check up to $n(n+1)$ replicas randomly. The peer itself probes n random replicas, and receives measured values from up to n different peers that have been able to probe n random replicas as well. Such a two-step approach produces lower communication costs: even where $n = 33$, and with 66 messages we are able to check up to 1112 replicas, and achieve good precision in a network of any size. This is a significant gain, because the initial idea requires 1068 messages in order to obtain the same precision. Algorithm 7 demonstrates the proposed method in the form of pseudo-code. Later evaluation shows that 98% of all measurements have an error rate lower than 0.03, proving that the applied strategy gives the results with expected error.

Better Accuracy under Churn

The proposed measurement technique produces accurate results, if peer availability is constant or changes slowly over time. In reality, the assumption cannot hold; peer availability varies significantly at run-time, and can depend on many parameters, such as user behavior, time of day, or content popularity. Periods of time when the number of online peers decreases more or less rapidly are defined as churns. During and some time them, peers measure much higher peer availability than the actual one, and do not create enough replicas immediately. However, as time passes, measurements become more precise, and peers create a sufficient number of replicas recovering the requested data availability.

Briefly testing the protocol under churns proves the validity of this statement: it is able to restore the given data availability after the system stabilizes again, but the recovery period is long. In a stable DHT averaging helps to reduce the variance of the calculated value, i.e. to

Algorithm 7 *measureP*: Measuring the actual peer availability by using a lower number of messages

Require: $n > 0, R > 0$

$\mathcal{S} = \{\}$
while $|\mathcal{S}| \leq n$ **do**
  $k \leftarrow$ randomly chosen key from the local storage
  $i \leftarrow$ random number between 1 and R
  $probeKey \leftarrow replicaKey(k, i)$
  $\mathcal{S} = \mathcal{S} \cup \{probeKey\}$
end while
$liveReplicas \leftarrow 0$
$sum \leftarrow 0$
for $key \in \mathcal{S}$ **do**
  $peerAvailabilityResponse \leftarrow route(key, peerAvailabilityRequest)$
  if $peerAvailabilityResponse.replicaExists$ **then**
    $liveReplicas \leftarrow liveReplicas + 1$
  end if
  $\mathcal{M} \leftarrow peerAvailabilityResponse.measurements$
  for $m \in \mathcal{M}$ **do**
    $sum \leftarrow sum + m$
  end for
end for
return $\frac{sum + \frac{liveReplicas}{n}}{n+1}$

get the value within the allowed error. During a churn, the approach does not produce precise measurements, because peers perform them at different points in time, when peer availability is different as well. Therefore, the calculated values contain a higher error and peers do not always create enough replicas immediately to compensate the effect of churn. However, as time goes by, DHT stabilizes again, and measured values are again within the defined error, making possible to restore fully the requested data availability.

Our goal is to make the protocol more reactive, i.e. recovery should be quicker, even nonexistent under weaker churn. Measured average peer availability should be as precise as possible, even during churns. As can be seen, just averaging received values is not a good way to achieve that, because it does not explore the information about the time when a measurement is made. Therefore, the main challenges are how to

- represent the measurement's time independently from the peer's clock
- limit the variance in the calculated peer availability during churns

Each measurement is accompanied with a time stamp. Peers do not have synchronized clocks and they are usually spread across multiple time-zones. Thus, the time stamps that are traveling with the measurements must be converted from the absolute time stamp into a relative one. A relative time stamp describes how many time units before the current moment an event has happened. For example, if an absolute time stamp is **"Jan 2 10:23:12 2006"** and the current time is **"Jan 2 12:33:30 2006"**, then the corresponding relative time stamp is **"-2:10:18"**, meaning that the given event has happened 2 hours, 10 minutes and 18 seconds previously.

Peers do not average the received values; they are placed in a history. After making n valid probes, the peer's history contains a number of measurements done in the past. A good estimation of the peer availability can be done by finding a curve that fits the measured data with the lowest error. Such a technique is called regression analysis [MS03] and its simplest form is known as linear regression analysis, where a linear curve $y = a + bx$ is fitted to the data. In our approach y is the measured peer availability p, and x is time t. Hence, our aim is to determine the curve

$$p(t) = a + bt \tag{4.26}$$

i.e. its coefficients a (value at $t = 0$) and b (the curve's slope). Since all measurements were made in the past, the received timestamps are negative. The present time is represented with $t = 0$, and the estimation of the current peer availability is then the value of $p(0)$.

Note that the methodology presented in this section allows fitting to any other type of curve, or fitting can be done using any optimization techniques. Finding the optimal fitting strategy (i.e. a curve with the smallest error) is out of the scope of this thesis, because it depends strongly on the application scenario.

There are plenty of techniques for computing the values of coefficients a and b, and one of them is the Least Squares method [MS03], which attempts to minimize the sum of the squares of the ordinate differences (called residuals) between points generated by the curve and corresponding points in the data. Supposing that the peer's history contains a set of N_h measurements (t_i, p_i) that should be approximated with a curve $p(t)$ using the Least Square method, the following sum

$$S = \sum_{i=1}^{N_h} (p_i - p(t_i))^2 \tag{4.27}$$

$$= \sum_{i=1}^{N_h} (p_i - a - bt_i)^2 \tag{4.28}$$

must be minimal. The sum has a minimum under the following conditions:

$$\frac{\partial S}{\partial a} = 0 \tag{4.29}$$

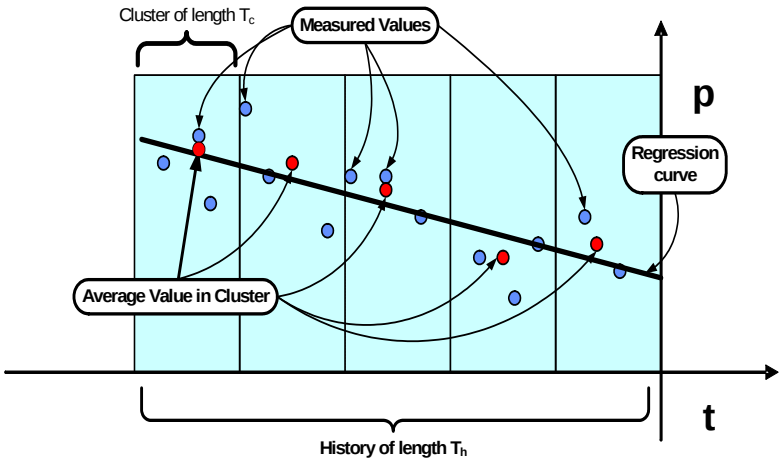


Figure 4.1: Visual representation of a peer's history on the timeline, its clustering, and fitting the curve to the average values in clusters

$$\frac{\partial S}{\partial b} = 0 \tag{4.30}$$

Solving this equation system provides solutions for a and b:

$$b = \frac{N_h \sum_{i=1}^{N_h} p_i t_i - \sum_{i=1}^{N_h} p_i \sum_{i=1}^{N_h} t_i}{N_h \sum_{i=1}^{N_h} t_i^2 - \sum_{i=1}^{N_h} t_i \sum_{i=1}^{N_h} t_i} \tag{4.31}$$

$$a = \frac{\sum_{i=1}^{N_h} p_i - b \sum_{i=1}^{N_h} t_i}{N_h} \tag{4.32}$$

The history size at every peer should keep only values of a limited age, so that the computed curve can closely represent the current situation in the DHT. If the history length increases, the computed curve would become more and more insensitive to churns as time passes. The value $p(0)$ would converge to the average peer availability in total, but peers would not be able to detect any increase or decrease of peer availability in longer periods of time and react properly.

The fitted curve is accurate only if data points have low variance. As demonstrated in Section 4.4, the low variance can be obtained by averaging the received values. Averaging values with the same timestamp would be optimal, but the chance that many measurements have been done at the very same moment is very low. Thus, in order to produce data points with lower

variance, a tradeoff must be made. The history is divided into clusters of defined durations, and every cluster is represented with a value - an average of all values within the clusters. The major assumption here is that during the time period equal to the cluster duration, the average peer availability changes insignificantly. Therefore, the average values in the cluster have a low variance, and can be used for fitting the curve. Figure 4.1 visualizes the whole process. It displays values stored in peer's history, how they are clustered, and the curve that fits the average values in clusters.

The clusters have been introduced due to lack of enough measurements with the same timestamp. We assume that peer availability does not change significantly within the duration of a cluster. In order to keep the fitted curve close to the actual value in the DHT, cluster length must remain small. Otherwise, the value that represents the cluster becomes more insensitive to churns, and the peers measure less precise results.

To summarize, here is a list of steps needed for computing the current average peer availability:

1. Init: history has a length of T_H time units, cluster length is equal to T_c time units, where $T_H = kT_c$ and $k \in \mathbb{N}$
2. From the set of locally stored replicas, and knowing the last needed number of replicas R, a peer generates randomly a number replica keys using Formula 4.1
3. Peer issues *PeerAvailabilityRequest* messages with generated keys until it gets back n valid probes.
4. Received *PeerAvailabilityResponse* messages correspond up to n different peers; received measurements are placed in the local history
5. Peer groups measurements into clusters, calculates the average values, and uses them in linear regression analysis
6. The computed average peer availability is equal to $p(0)$, where $p(t) = a + bt$

The pseudo-code is shown in Algorithm 8. The measurement procedure defined allows the obtained values to be closer to the actual ones. As explained at the beginning of this Section, just averaging individual measurements would not give a good estimation of the peer availability during churn (i.e. the period when the peer availability decreases). Eventually, the measurements will become precise again, but until this happens, peers will believe that the peer availability is higher than it actually is.

4.5 Adjusting Number of Replicas

We now have all the pieces needed to make our protocol self-adaptable, i.e. to achieve and maintain the requested data availability and consistency even if peer availability changes. This

Algorithm 8 $measureP_{regression}$: Measuring the actual peer availability via building the regression curve

Require: $n > 0$, $R > 0$, $T_H = kT_c$, $k \in \mathbb{N}$

$\mathcal{S} = \{\}$, $\mathcal{V} = \{\}$
while $|\mathcal{V}| \leq n$ **do**
  $k \leftarrow$ randomly chosen key from the local storage
  $i \leftarrow$ random number between 1 and R
  $probeKey \leftarrow replicaKey(k, i)$
  if $probeKey \notin \mathcal{S}$ **then**
    $\mathcal{S} = \mathcal{S} \cup \{probeKey\}$
    $peerAvailabilityResponse \leftarrow route(probeKey, peerAvailabilityRequest)$
    $v \leftarrow getLocalReplicaVersion(probeKey)$
    if $peerAvailabilityResponse.replicaExists$ **then**
      if $peerAvailabilityResponse.replicaVersion = v$ **then**
        $\mathcal{V} = \mathcal{V} \cup \{probeKey\}$
        $liveReplicas \leftarrow liveReplicas + 1$
      end if
    else
      $\mathcal{V} = \mathcal{V} \cup \{probeKey\}$
    end if
    $\mathcal{M} \leftarrow peerAvailabilityResponse.measurements$
    for $m \in \mathcal{M}$ **do**
      if m younger than T_H **then**
        put m in *history*
      end if
    end for
  end if
end while
put $\frac{liveReplicas}{n}$ in *history*
divide *history* into k clusters
compute the cooeficients a and b from $p = a + bt$
return $p(0)$

can be done if managed data are replicated a number of times as defined by Formula 4.8. It depends on the actual peer availability p, and since it changes over time, the number of replicas for already stored and newly created objects must be adjusted, in order to keep data availability at the requested level.

Joining the DHT for the first time, a peer gets information about the number of replicas needed

Algorithm 9 *adjustNumberOfReplicas*: Adjusting the number of replicas of data locally managed by a peer

Require: $a > 0$
1: $R' \leftarrow R$
2: $R \leftarrow measureP_{regression}$
3: **for** all *key* in local storage **do**
4: $t \leftarrow getFromStorage(key)$
5: **if** $t.ron > R$ **then**
6: $removeFromStorage(key)$
7: **end if**
8: **if** $R > R'$ **then**
9: **for** $i = R' + 1$ to R **do**
10: $rk \leftarrow replicaKey(t.Key, i)$
11: $t.ron \leftarrow i$
12: $store(rk, t)$
13: **end for**
14: **end if**
15: **end for**

by reading it from a config file or asking neighbors. Later on, every peer measures the current peer availability at least once per online session, and iterates over replicas managed in its storage. By knowing the requested data availability, the peer is able to calculate the new value for the number of replicas R (Formula 4.8). It is compared with the previous known number of replicas R', and the peer does one of the following:

- $R > R'$: creates new replicas of data managed in the local storage. They will be stored under the keys $replicaKey(Key, j), j = R' + 1, \ldots, R$ elsewhere in the DHT.
- $R < R'$: fewer replicas are needed than before; if the replicas with the ordinary number *ron* greater than R are managed in the local storage, they will be removed.

Algorithm 9 shows the pseudo-code. Storing new or reading existing objects takes into account the last measured peer availability in order to determine the appropriate number of replicas used then by $store_{immutable}$/$store_{mutable}$ and $lookup_{immutable}$/$lookup_{mutable}$ operations.

4.6 Costs

In general, the total costs of a replication protocol consist of two parts: communication and storage costs. Our protocol is self-adaptable; it tries to reach and keep the requested data availability with minimum storage overhead at any point in time. If the initial number of replicas is not

sufficient to ensure the requested data availability, new replicas will be created, i.e. storage costs will be increased. On the other hand, if there are more replicas than needed, peers will remove some of them, reducing storage costs, but preserving the availability of data.

4.6.1 Immutable Data

Let us denote with S the average storage costs per peer in a DHT built by N peers that are online with probability p. If it manages M objects, the minimum of storage overhead per peer S_{min} that delivers the requested data availability a is:

$$S_{min} = \frac{M}{N}R \tag{4.33}$$

where R is computed as in Formula 4.5.

Every $store_{immutable}$ generates R *StoreRequest*. A $lookup_{immutable}$ does not have fixed costs, it stops sending *LookupRequest* messages when a replica is found. Sometimes the 1^{st} replica is already available. In an extreme case, R messages must be generated in order to figure out if the requested object is available or not. It can be shown that on average a $lookup_{immutable}$ generates $\frac{R}{2}$ messages, before finding a replica of an object.

The proposed approach introduces additional communication costs generated by measuring peer availability p and adjusting the number of replicas. Every measurement generates n *PeerAvailabilityRequest*, where n is the number of replicas to probe. As stated in section 4.4, n depends on the absolute error δ we want to allow, and the probability that the measured value is within the given interval $(p-\delta, p+\delta)$. If the actual number of replicas R is greater than the previous one R', the peer will additionally create $(R-R')\frac{M}{N}$ *StoreRequest* messages on average.

Due to changes in the DHT topology, some replicas can be moved to peers where they belong according to the current topology. The number of moves, i.e. the number of messages generated by replica moving strongly depends on the peer offline/online rate and the underlying DHT implementation. Its analytical estimation is beyond the scope of this thesis, but it should be measured during the evaluation.

4.6.2 Mutable Data

As indicated in Section 4.3.2, additional replicas are created as the result of an update. The first update reaches pR replicas on average, and additionally inserts $R(1-p)$ replicas within the DHT. Afterwards, the total number of replicas of a given object R_t is equal to $R(2-p)$ on average. The same happens when an object is updated for the second time, and the total number of replicas increases to $R(2-p)^2$. Following the same pattern, it can be shown that after u updates, an object can have up to

$$R_t = R(2-p)^u \tag{4.34}$$

replicas on average in the DHT.

Fortunately, due to the DHT routing mechanism, R_t does not grow indefinitely. Namely, all R_t replicas are stored in the DHT using only R different keys. Therefore, even if more than R replicas are online, only R at most are accessible (i.e. $pR_t \leq R$). Hence, the maximum number of replicas (online+offline) R_t^{max} that can be managed in the DHT for a given object is when $pR_t^{max} = R$. An update request is able to address all R replicas managed under different keys, not inserting any new replica version into the DHT. Thus:

$$R_t^{max} = \frac{R}{p} \tag{4.35}$$

It is reached when an object is updated at least

$$u_{sat} = \left\lceil \frac{-\log p}{\log(2-p)} \right\rceil \tag{4.36}$$

times.

Due to an arbitrary number of updates performed on every object, they might have a different number of replicas, and the average storage costs per peer S can be expressed as

$$S = \frac{\sum_{i=1}^{M} R_t^{(i)}}{N} \tag{4.37}$$

where $R_t^{(i)}$ represents the total number of replicas (online+offline) for i^{th} managed object.

The average storage costs S are at the lowest level if the DHT manages immutable data ($R_t = R$), and then the formula becomes identical to Formula 4.33. The upper bound is when all objects are updated at least u_{sat} times:

$$S_{max} = \frac{MR}{pN} \tag{4.38}$$

Therefore, managing mutable data can require up to p^{-1} times more space to keep the requested guarantees on the same level as when data are not subject to any update. For example, in a DHT with 20% peer availability, managing mutable data could take up to 5 times more space, whereas peer availability is 50%, the storage space used per peer is only twice as high.

As before, inserting or updating an object generates R *StoreRequest* messages. Before introducing updates, $lookup_{immutable}$ terminates by finding any available replicas. On average, this happens after $\frac{R}{2}$ *LookupRequest* messages. Knowing that objects are potentially mutable increases the costs of $lookup_{mutable}$. It tries to find all R replicas and returns the one with highest version number. Thus, enabling updates generates twice as many messages by every $lookup_{mutable}$ on average.

On other hand, measuring peer availability does not depend on the nature of managed data. Thus, the related costs remain the same as described in Section 4.6.1.

4.7 Summary

This chapter has described in detail the replication protocol that provides transparently an arbitrary data availability and consistency guarantees in a dynamic, arbitrarily available DHT. At the same time, it does not exploit any knowledge on data access pattern or peer behavior.

Stored replicas can be found when needed only if their location is known. A usual place for managing mappings between object IDs and locations where they are replicated, is the replica directory. Since DHTs are fully decentralized, the state-of-the-art approach cannot be applied. Instead, we have built the decentralized replica directory on top of a DHT itself. Thanks to the DHT properties, the keys used for storing the replicas are at the same time their locations. The mappings between an object key and the replica locations (the replica keys) have been realized via a globally known function that hashes a combination of object key and the replica ordinary number. Using a good hash function like MD5 or SHA-1 guarantees that all replica keys are practically unique. Also, all replica keys of an object are quite distant from each other in the key space.

Maintaining the requested data availability and consistency in a self-adaptable way requires peer to be able to add new or remove unneeded replicas of an object. The keys of additional replicas must be generated using the same global function. Thus, the object key and the replica ordinary number should be attached to a replica value managed within the DHT.

After designing the decentralized replica directory, a protocol suitable for guaranteeing high availability of immutable data was presented. The guarantee has been achieved by replicating data a number of times, hoping that at least one of the replicas will be available when an access is required. The analysis carried out has shown that the number of replicas exponentially depends on the requested data availability a, and the actual peer availability p. In order to be self-adaptable, the main challenge is how to measure the actual peer availability.

If mutable data should be kept highly available and consistent, consistency becomes the main issue. We have adopted a weak consistency model that, unlike existing approaches, provides guarantees about the minimal consistency level until all replicas are eventually updated. An update overwrites all replicas available at that moment, whereas the updated value of offline replicas will be inserted in the DHT. When an object is requested, the protocol must return the freshest available replica. To distinguish old from new replica values, a version number and timestamp are attached to the managed replica value. Additionally, the version number helps to detect concurrent updates on the same object if replicas are updated in a predefined order. The analysis of the proposed consistency model has shown the relationship between the necessary number of replicas, the requested level of data availability and consistency, and the actual peer availability. Surprisingly, the number of needed replicas is the same as when the DHT manages immutable data.

Next, Section 4.4 presented how to measure the current peer availability with high precision, even without any knowledge of the underlying DHT topology. Thanks to the properties of the defined decentralized replica directory, peer availability can be computed by detecting replica availability. Of course, we are not in a position to probe all the replicas stored in the system.

In order find how many probes needs to be made in order to get the measurement with an acceptable error rate, we have based our calculations on the usage of confidence interval theory. Besides probing itself, a peer collects the measurements made by the others at a different point in time. The actual peer availability is then computed by fitting a linear curve on the collected measurement by using the Least Square method.

Solving all the above issues, the peers are ready to adjust the number of replica of locally stored objects when the peer availability changes. They iterate over replicas stored in their local storages and calculate the new number of replicas for the given object, taking into account the requested data availability and consistency and the measured peer availability. If it is higher than before, the additional replicas will be created. On the other hand, if the current situation delivers the same data availability and consistency with fewer replicas, those not needed will be removed from peers' storages.

Finally, an estimation of the protocol's overhead was given. Obviously, replicated data require more storage space. By looking at the communication side, the protocol generates more messages when storing or retrieving an object. In addition to that, measuring peer availability produces a fixed number of messages. The costs of managing mutable and immutable data have been compared as well. If the same guarantees should be maintained, it turns out that the mutable data can generate, in an extreme case, up to p^{-1} times higher storage costs, where p is the actual peer availability. Usually, the update rate is much lower, making the ratio closer to one. As regards communication costs, inserting/modifying an object requires the same number of both cases, whereas lookups generate twice as many messages when the DHT manages mutable data.

Chapter 5

Evaluation

THE protocol designed for the thesis is evaluated in this chapter. Its behavior is tested within various scenarios that check the goals described in Section 1.2. The results are obtained using a custom-made simulator, described in Section 5.1. It is highly configurable, allows the running of many different types of scenarios, and collecting various kinds of information needed to evaluate the protocol's properties and performance. In order to test only relevant cases, Section 5.2 defines the values of parameters that should be fixed for all scenarios described in Section 5.3. Section 5.4 presents a set of criteria used for checking if the protocol fulfills the planned goals.

The results obtained by managing the requested guarantees on immutable data are presented in Section 5.5. A smaller portion of results has already been published in [KWR06a, KWR06b]. Section 5.6 contains the graphs obtained by running our protocol on mutable data, and is an extended version of results presented in the paper [KWR07]. All graphs are discussed, and the outcomes of the same scenario on mutable and immutable data are compared. Also, the average storage costs are compared with the theoretical values described in Section 4.6.

5.1 Simulator

The results were obtained using a custom-made simulator. This was necessary although several peer-to-peer simulators such as PeerSim [JJMV06], P2PSim [GKL+05a], GPS [YAG05], 3LS [TD03], or OverSim [GKL+05b] are available. Unfortunately, we have found none of them to be suitable for the planned evaluation. Our goal was to use the same code for simulations and later on for real deployment. The protocol is implemented as a wrapper around FreePastry [Uni06], an open-source DHT Pastry [RD01a] implementation written in Java. Due to their non-Java implementations, P2PSim, GPS, and OverSim were not good candidates. The rest of the Java-based simulators required non-trivial changes to the implementation of the protocol in order to run it within the simulator.

The simulator developed executes the same code that is ready to be deployed in a real environment, i.e. the implementation is completely unaware of the existence of the simulator. The simulator itself acts as the application to the established DHT and uses a common DHT API [DZDS03] for invoking DHT functions.

Every simulation consists of the following steps:

1. DHT creation – a number of peers are deployed.

2. Populating the DHT with a number of objects. They are inserted using randomly chosen peers, and replicated transparently using the implemented protocol. The initial number of replicas is obtained from a config file.

3. Running a scenario.

The simulator has a discrete notion of time. Every scenario runs a number of time units, and during each of them, the following actions are performed:

1. Changing DHT topology

   Peers whose online session is over go offline, and some other peers with the probability p come back. Session length is generated by using Poisson distribution, with the given average session length λ. In order to simulate churns, the simulator can increase or decrease linearly the peer availability p over a given period. The duration of this process, its start and stop value, and triggering mechanism is configurable.

   Each time a peer comes online, it measures the actual peer availability, determines the number of required replicas, and adjusts the number of replicas for locally stored data according to the mechanism presented in Section 4.4.

2. Measuring actual data availability and consistency

   The simulator measures the actual data availability and consistency delivered by our protocol by trying to access all objects stored at the beginning of the simulation. It iterates over keys of previously stored objects, picks up a random online peer, which then issues a *lookup*. The simulator tracks the latest values of all managed objects, and therefore is able to determine if the value found is up-to-date.

3. Optionally: updating available objects according to a defined distribution.

   The simulator can update managed objects according to an arbitrary uniform distribution. Although such an update pattern is quite artificial, and rarely occurs in real deployment, it is a very good stress test for our protocol. If it behaves well under such conditions, then it will provide good performance as well, given any other more realistic update pattern.

5.2 Settings

The main configuration parameters of the simulator are the number of peers N that form the DHT, the requested data availability a, the average peer online probability p, the average session time λ, the initial number of replicas R, the number of replicas to probe n, the history and cluster length.

These parameters offer a huge number of possible scenarios to simulate, but only those that fit the problem described in Section 1.2 will be chosen. Delivering high data availability counts only in real-world applications, and thus the simulations take into account only 99% data availability.

Our protocol does not specify any particular requirement on peer availability in DHT, and therefore it should be evaluated both in DHTs with low and high peer availability. Typical examples of real peer-to-peer networks with low peer availability (about 20%) are file-sharing networks like KaZaA, Gnutella, or eDonkey. On the other hand, the average peer availability in the BRICKS community should not go below 50%, and it can be considered as high.

DHT size N has been fixed at 400 peers, the average session time λ at three time units, and the number of replicas to probe n at 30. As stated in Section 4.4, such a number of probes produces the measurements with a precision of ± 0.03 in an arbitrary large P2P network. In order to be able to react fairly quickly to churns, and yet to still have enough data points to build the regression curve, history length is set to 15 time units, and it is clustered into segments of 3 time units.

A major assumption built into the proposed protocol is that all replicas of an object are stored on different peers within the network. It mainly influences the precision of the measured peer availability. If this assumption is not fulfilled, the correlation of peer online probability and replica availability is corrupted, thereby influencing precision. As stated in Section 4.1, this assumption holds in a fairly large network.

Thus, the size of DHT used in the simulations has not been chosen randomly. In particular, our aim was to find a network where the above described effect does not occur. Thus, a number of simulations has been performed in order to track errors in the measurements obtained. The experiments have been done in low-availability[1] DHTs of various sizes, managing immutable data. The rate of measurement error drops with increasing network size, and it already stabilizes for networks with more than 300 peers. To be on the safe side, our simulations should be executed in a network with at least 400 peers.

Getting accurate measurements requires that peers have enough replicas to be able to choose 30 replica keys at random. Thus, peers should have enough replicas in their storage. DHTs are populated with $M = 6000$ objects, and if the replication factor is greater than 1 ($R > 1$), every peer owns more than 30 replicas on average. Hence, a peer is able to select enough keys for probing, and to obtain a good estimation of the actual peer availability afterwards.

[1] The peer online probability of 20% has been selected as the lowest probability used in this evaluation and therefore has been applied consequently on the determination of the network size. This probability sounds reasonable as compared with the probabilities reported in [SGG02b].

Parameter	Value
DHT size N	400
Number of managed object M	6000
Online session duration λ	3
Requested data availability a	99%
Update rate u	10%
Number of replica probes n	30
History length	15 time units
Cluster length	3 time units
Simulation length	200 time units
Number of runs per scenario	10

Table 5.1: Set of fixed simulator parameters, used throughout the evaluation

The chosen update distribution is uniform, and each of the managed objects can be updated at any time with a probability of 10%. This is quite high and unlikely to happen in real-world deployment. The BRICKS decentralized storage manages various metadata such as service or collection descriptions. Although mutable, they do not change so often, i.e. their update probability is much lower than 10%. However, the selected update distribution is good for carrying out a stressful evaluation. Since our simulations last 200 time units, approximately 120000 updates are performed during every simulation. If the protocol behaves well after so many updates, it will perform at least similarly or better in a real-world deployment.

Table 5.1 summarizes values of all parameters that are fixed throughout the evaluation.

5.3 Scenarios

The next step is to define relevant scenarios, where the thesis goals stated in Section 1.2 can be tested:

- **Ability to reach** the requested average data availability without knowing peer availability, nor data update distribution *a priori*

  A DHT with a stable average peer availability over time is created and data are initially replicated in a number of replicas and this does not guarantee the requested data availability. Our protocol should be able to detect this, and increase the number of replicas until the data are available with the requested probability.

- **Ability to maintain/recover** the requested data availability without knowing peer availability, nor data update distribution *a priori*

	initial peer availability	Churn		Replica number		checks if availability and consistency are
		type	duration	initial	needed	
1	20%	-	-	5	21	reached
2	50%	-	-	3	7	reached
3	50%	-	-	11	7	maintained while adjusting storage costs
4	20%	-	-	30	21	maintained while adjusting storage costs
5	50%	50% → 20%	15	7	21	recovered
6	50%	50% → 20%	100	7	21	recovered
7	20%	20% → 50%	15	21	7	maintained while adjusting storage costs
8	20%	20% → 50%	100	21	7	maintained while adjusting storage costs

Table 5.2: Scenarios used for evaluating the protocol

This can be validated by running scenarios where a stable DHT experiences a churn, i.e. the number of online peers starts to decrease. The previously determined number of replicas cannot deliver the requested guarantees any more. The churn can be shorter or longer, but after peer availability stabilizes again, the requested data availability must be recovered within a short period.

- **Ability to adjust storage costs**

  Data should be replicated initially more than is actually needed to maintain the requested data availability. The goal is fulfilled if such a situation is detected and unnecessary replicas are removed.

  Unneeded replicas also appear in "negative" churns, i.e. when more peers come online during some periods, and consequently peer availability increases. The number of replicas has been adjusted to the lower peer availability. However, more peers are online now, and the number of replicas becomes too high for the new peer availability. Again, the protocol behaves well if the storage costs are reduced close to the needed minimum.

Based on the goals that our protocol should fulfill, a set of scenarios has been defined. Table 5.2 summarizes their properties: how peer availability changes over time, what the initial replication factor is, how many replicas are actually needed to guarantee 99% data availability, and which particular goals are checked.

5.4 Criteria

The results obtained are evaluated by constructing the following graphs:

- average, maximal, and minimal data availability obtained during simulation time
- average error obtained by reaching the requested data availability

  An error value at the time unit *t* represents the average absolute error within the last 20 time units of simulation. When the required data availability is achieved, the error should become low and remain at such a level until the end of the simulation. Obviously, the longer the period from which the error is computed, the closer its value will be to the average value achieved throughout the whole simulation. However, applications could have different requirements on the obtained error during some intervals, and therefore choosing the suitable window length needed to compute the error is application specific. Unfortunately, such precise requirements were not available during the evaluation.

- average, maximal, and minimal storage costs achieved per peer, compared with the theoretical prediction represented by Formula 4.5 or 4.8

In addition to this, we observe generated communication costs. The simulator is able to track the total number of *PeerAvailabilityRequest* and *StoreRequest* messages per time unit, generated during peer availability measurements and replica adjustments. The measurement costs should be stable over time, whereas the number of *StoreRequest* messages should decrease when the requested data availability is reached.

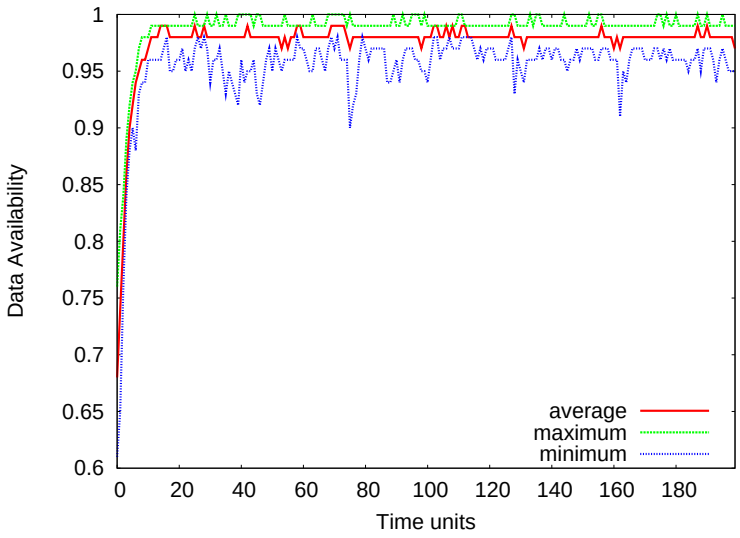
5.5 Managing Guarantees on Immutable Data

The scenarios defined in Section 5.3 are executed first on immutable data. Every simulation begins by storing all the objects in a DHT. After that, DHTs equipped with our protocol work on managing the data availability at the requested level. In the remainder of this Section, we check if our protocol delivers the requested guarantees according to the criteria defined and the logs collected from the simulation runs.

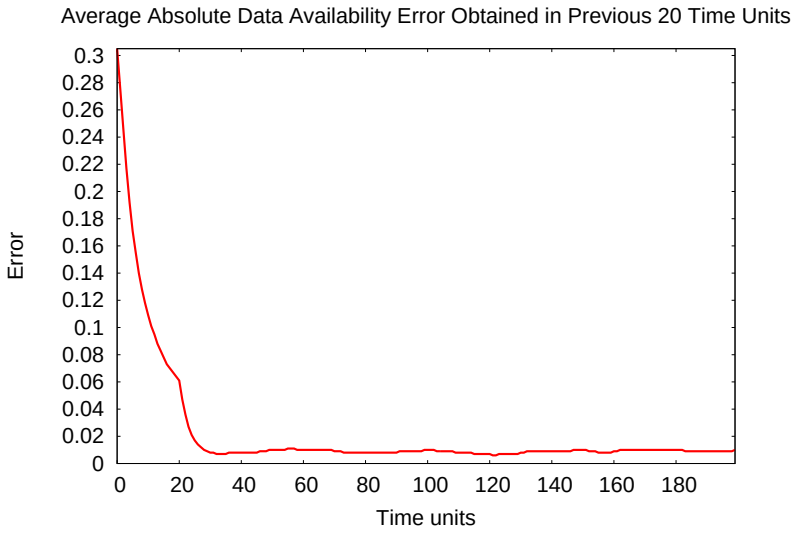
5.5.1 Stable Peer Availability

Reaching Data Availability

As defined in Scenario 1 and 2 (Table 5.2), objects are replicated initially fewer times than actually needed to guaranteeing the requested data availability of 99%. In the case of a low-availability DHT (Figure 5.1), every stored object is replicated 5 times. As Formula 4.5 defines, every object should have at least 21 replicas in order to maintain an availability of 99%.



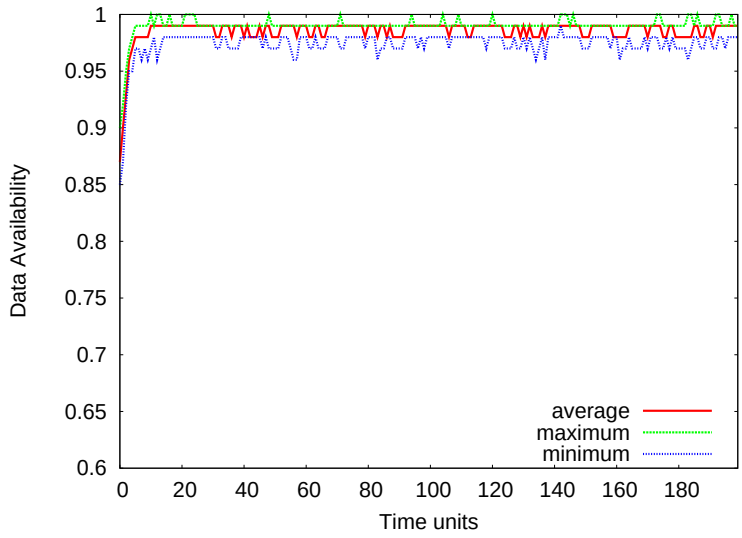
(a) obtained data availability



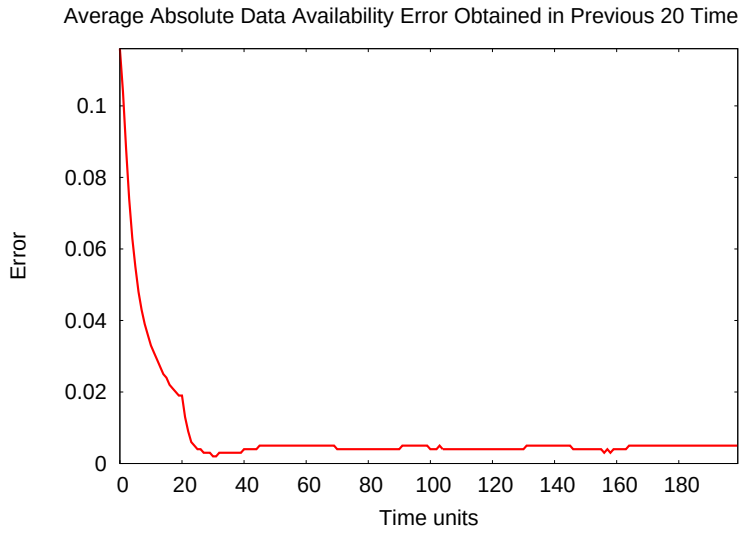
(b) average error in past 20 time units

Figure 5.1: Scenario 1: low-availability DHT with peer availability of 20%. Immutable data are replicated initially 5 times, but at least 21 replicas are needed for 99% data availability

Similarly, Figure 5.2 shows results obtained in the highly-available DHT, where peer availability is 50%, and the initial replication factor is 3. As in the previous cases, objects should be replicated at least 7 times in order to deliver the guarantees.



(a) obtained data availability



(b) average error in past 20 time units

Figure 5.2: Scenario 2: highly-available DHT with peer availability of 50%. Immutable data are replicated initially 3 times, but at least 7 replicas are needed for 99% data availability

Every scenario was executed 10 times and the values obtained have been used to calculate the average ("average curve"), the maximal ("maximum curve"), and the minimal ("minimum curve") achieved data availability across all runs.

Although data were initially replicated fewer times than actually needed, both DHTs enriched with our protocol were able to detect this, create more replicas of all managed objects and reach and maintain a level of data availability very close to the requested one. Figure 5.1b and 5.2b show the development of the average errors during the simulations. As it can be seen, the delivered data availability is slightly below the requested one, i.e. the error is around 0.01. In both cases, it is already reached after 30 time units.

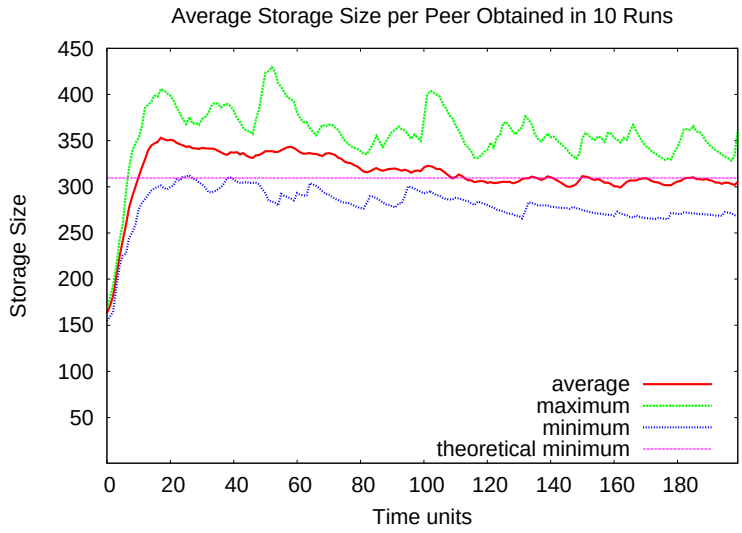
As it can be seen, the minimum curve on Figure 5.1a is not so smooth as the other two. The largest two outliers lay around time unit 80 and 160. As explained before, the curve represents the minimum of data availability at the given time unit obtained across 10 runs. The scenario simulates a DHT with the average peer availability of 20%. The actual value at a given time unit can vary around the average one. This can have an impact on the actual data availability, especially if the peer availability is low. For example, if the number of replica is determined for the peer availability of 20%, but in the next time unit it drops to 14%, the actual data availability would drop to 0.94. If the peer availability is 50% (Figure 5.2a), the variations of the actual peer availability do not produce such a big impact on the actual data availability. Hence, the outliers on the minimum curve are much lower.

As already discussed in Section 4.6, guaranteeing the data availability generates higher storage and communication costs. Both DHTs do not deliver the proper data availability from the beginning, and therefore more replicas are created, until the requested availability is reached. From the costs perspective, the average storage size per peer increases, and when the data availability is achieved finally, remains close to the value suggested by Formula 4.33. The increase of storage size can be followed also by looking at the number of generated *StoreRequest* messages. Again, after reaching the data availability, the number of *StoreRequest* messages should drop significantly.

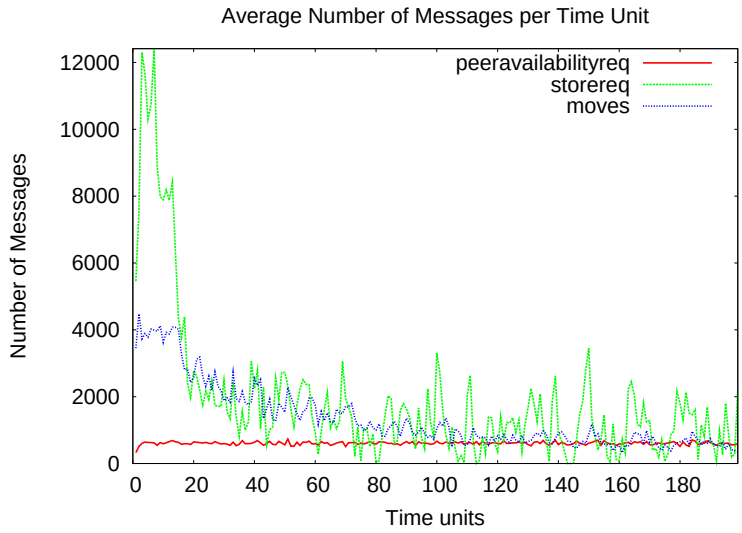
Figure 5.3 and 5.4 present the evolution of storage cost throughout the simulations. They plot three curves: the average (the "average" curve), the maximal (the "maximum" curve), and the minimal costs (the "minimum" curve) obtained within 10 runs. The curve "theoretical minimum" corresponds to the storage overhead predicted by Formula 4.33 and it is used for estimating how close the actual storage costs are .

As expected, the absolute overhead is higher in the low-availability DHT. Nevertheless, the pattern is the same as in the highly-available DHT. As soon as the data availability is reached, the average storage costs become close to the theoretical value. For Scenario 2, the average storage costs are a bit below the theoretical value. Replacing the chosen values for a and p in Formula 4.5, the exact number of replicas is 6.7. Of course, in reality, it must be rounded to the next integer. However, due to errors in measuring the peer availability, some peers could calculate that six (6) replicas are enough. The portion of such peers is not high, otherwise the delivered data availability will not be close to the requested one. However, the average number of replicas will be a bit under 7 replicas, and hence the average storage costs are a bit below the theoretical curve.

The increase of peer storage size can be observed also via "storereq" curve on Figure 5.3b and 5.4b. It displays the total number of generated messages in time, averaged over all runs. As



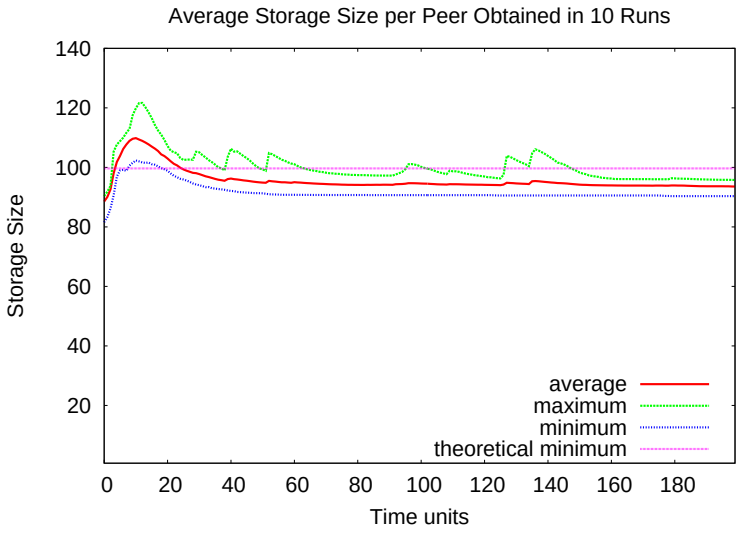
(a) average storage size per peer



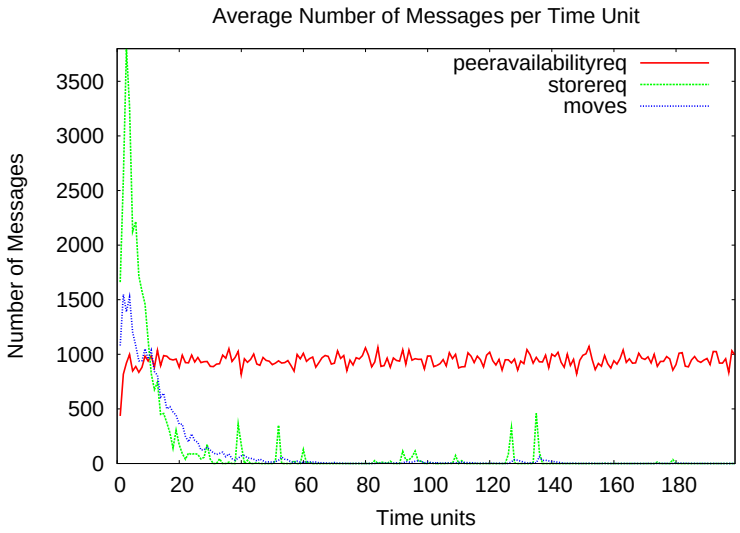
(b) communication costs

Figure 5.3: Scenario 1: generated costs for low-availability DHT with peer availability of 20%. Immutable data are replicated initially 5 times, but at least 21 replicas are needed for 99% data availability

soon as the data availability is very close to the requested one, the number of generated messages decreases significantly. Later on, only some sporadic peaks appear due to the error in measuring



(a) average storage size per peer



(b) communication costs

Figure 5.4: Scenario 2: generated costs for highly-available DHT with the peer availability of 50%. Immutable data are replicated initially 3 times, but at least 7 replicas are needed for 99% data availability

peer availability. Namely, the number of needed replicas R (Formula 4.5) depends exponentially on the measured peer availability p, and thus even a small measurement error can cause a change

in computed *R*, forcing peers to generate more *StoreRequest* messages, and consequently more replicas. For example, if the previous measured peer availability is 20%, but a peer measures now 18%, the number of needed replicas increases from 21 to 23, and two *StorageRequest* for every managed replicas will be generated. Additionally, the change of peer availability comes from the simulator itself. The simulator guarantees that the average peer availability has the requested value, but the actual value at every time unit varies around the average one. Therefore, the actual peer availability can be 18% sometimes and *StoreRequest* messages will be issued with a good reason.

The "peeravailabilityreq" curve presents the communication cost related the measurement of peer availability. It is practically constant over time. Comparing two DHTs, they are a bit higher in the highly-available DHT, because the peer online rate is higher, i.e. more peers come online at every time unit and measure the peer availability afterwards.

After adjusting the number of replicas for locally managed objects, a peer checks which replicas are not under its responsibility anymore. Every time when the DHT topology changes, some peers become responsible for smaller or larger part of the keyspace. All replicas that are not under peer's responsibility should be moved to proper destination peers – otherwise, they become unreachable, since the DHT routing mechanism sends requests elsewhere. As already stated in Section 4.6, the traffic generated by moving replicas from one peer to another cannot be modeled in a general case – details about used DHT implementation and peers behavior must be known.

The "moves" curve displays how many replicas are moved due to DHT topology changes. The obtained results shows what is the nature of this traffic. In the beginning, the data availability is bellow the requested one. Thus, all peers create a lot of new replicas that are stored on the currently available peers. However, these replicas should be placed eventually on peers that have the IDs closest to their keys. Therefore, a lot of moves happens immediately after replica creation, because they are moved towards their final destination. Meanwhile, the data availability is reached and new replicas are not created anymore. The remaining traffic corresponds only to the DHT topology change. Obviously, the topology can and does change more often when the peer availability is low. As Figure 5.3b demonstrates, it stabilizes around 1000 messages per time unit. It means that the routing rules change very often in Pastry when the peer availability is 20%. Contrary, if the peer availability is 50%, the routing rules are quite stable, and the "moves" curve decreases practically to zero (Figure 5.4b).

Maintaining Data Availability

Scenario 3 and 4 (Table 5.2) define a situation when date are replicated initially more times than actually needed for guaranteeing 99% of data availability. Our protocol should be able to detect this and to remove unnecessary replicas, while maintaining the requested data availability with a low error.

As Figure 5.5 and 5.6 show, the data availability is maintained all the time without any significant error. At the same time, the average storage costs decrease to a level close to the theoretical minimum (Figure 5.7a).

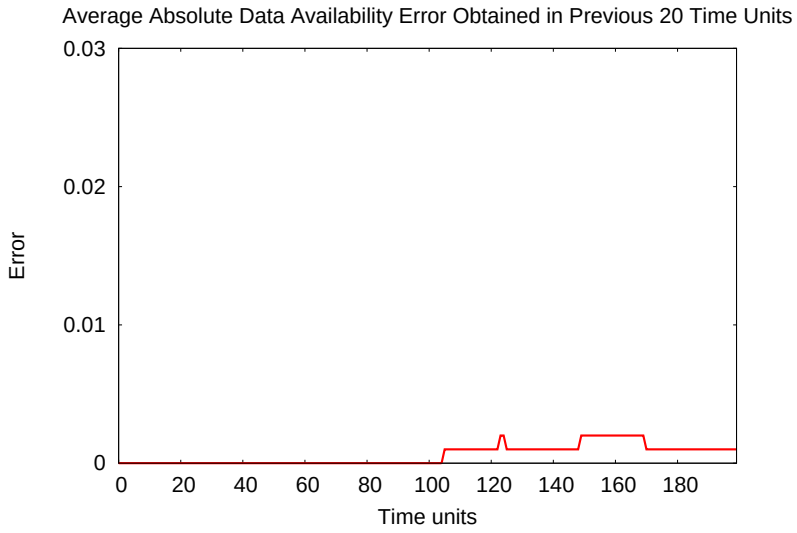


Figure 5.5: Scenario 3: highly-available DHT with the peer availability of 50%. Immutable data are replicated initially 11 times, but 7 would be sufficient to manage 99% data availability

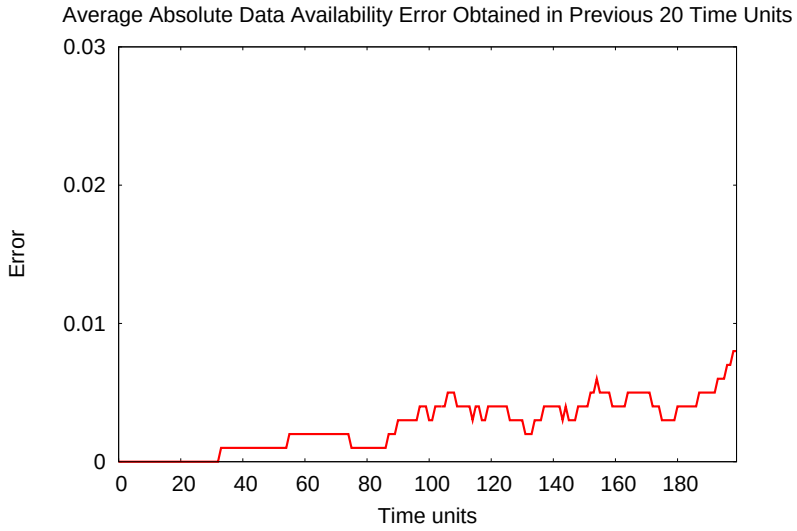


Figure 5.6: Scenario 4: low-availability DHT with the peer availability of 20%. Immutable data are replicated initially 30 times, but 21 would be sufficient to manage 99% data availability

Since no new replicas should be created, the number of *StoreRequest* messages is very low (Figure 5.7b). From time to time, some messages are still generated due to peer availability measurement error, but the number of messages is pretty low (not more than 200 in the network of 400 peers). The highly-available DHT has a quite stable topology, i.e. the routing rules changes slowly. Thus, the number of moved replicas is very low as well. As before, the measurement of the peer availability is independent from the initial replication factor, or the peer availability. The costs remain on the same level as on Figure 5.4b.

As discussed before, an error in measuring the peer availability influences the computed number of replicas significantly in a low-availability DHT. Consequently, peers create unnecessary more replicas and generate higher traffic. This is what happens even we have more replicas than needed (Figure 5.8b). However, compared with Figure 5.3b, some commonalities can be observed. Namely, the number of *StoreRequest* messages follows the same pattern as in Scenarios 1 and 2, after reaching the requested data availability.

Conclusion

The previous evaluation shows that the protocol presented in this thesis is able to reach the requested data availability and to maintain it afterwards with an insignificant error and the storage overhead very close to the predicted minimum. If peer availability is underestimated, and therefore data are replicated initially more times than actually needed, our protocol detects this and reduces the number of replicas while keeping data availability at the requested level.

The clear advantage of a DHT equipped with the protocol presented is deployment in an environment whose parameters such as peer availability are not well-known or cannot be predicted well. The protocol itself will add needed/remove unneeded replicas, thereby delivering the defined guarantees.

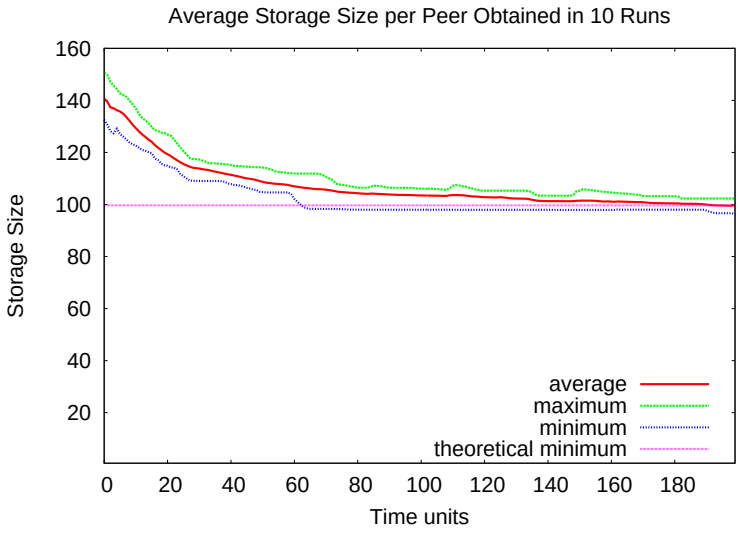
5.5.2 DHTs during Churn

Scenarios 1-4 are a good testbed for the protocol's basic properties: ability to reach and maintain the requested data availability, when data are initially replicated less or more than needed, and peer availability is stable. As the results presented in the previous Section show, the goals defined in Section 1.2 have been successfully achieved. However, the scenarios used are too idealistic – in reality, peer availability is not stable over time, and can sometimes fluctuate strongly.

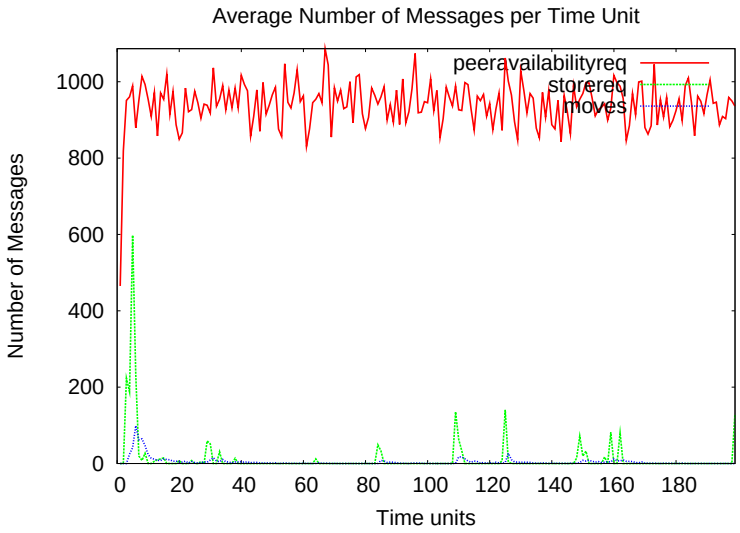
Recovering Data Availability

The following experiments evaluate the influence of a churn on our replication protocol, i.e. its ability to recover or maintain the requested data availability after or during periods of unstable peer availability.

The highly-available DHT equipped with our protocol experiences a high rate of churn (Scenario 5): the peer availability drops linearly from 50% to 20% during 15 time units. After that, it



(a) average storage size per peer

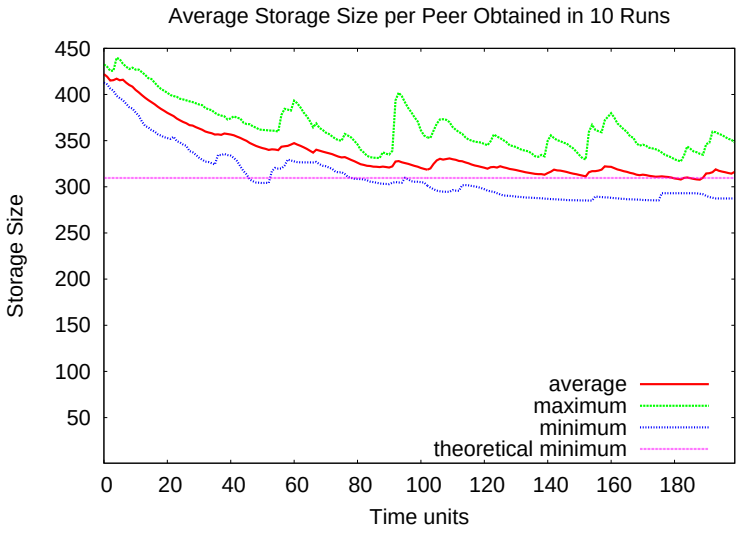


(b) communication costs

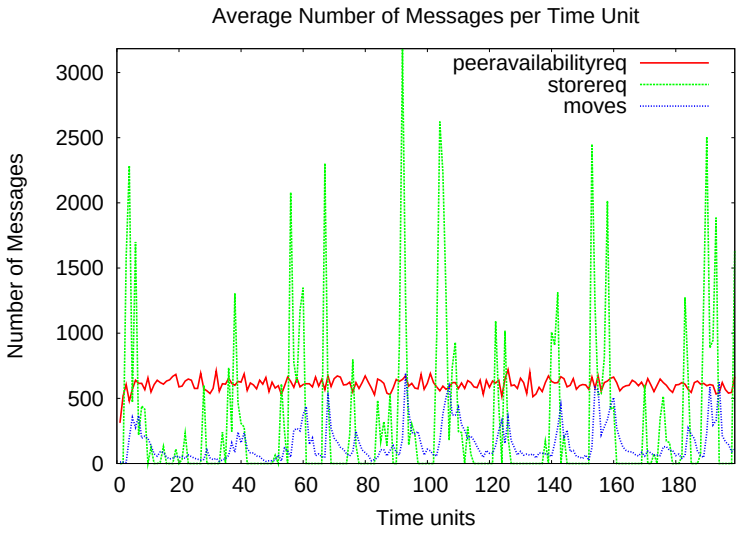
Figure 5.7: Scenario 3: generated costs for highly-available DHT with the peer availability of 50%. Immutable data are replicated initially 11 times

remains at 20%. Before the churn, it was stable as well, and the requested data availability was being delivered.

Figure 5.9 demonstrates that we are able to recover data availability shortly after the churn is



(a) average storage size per peer

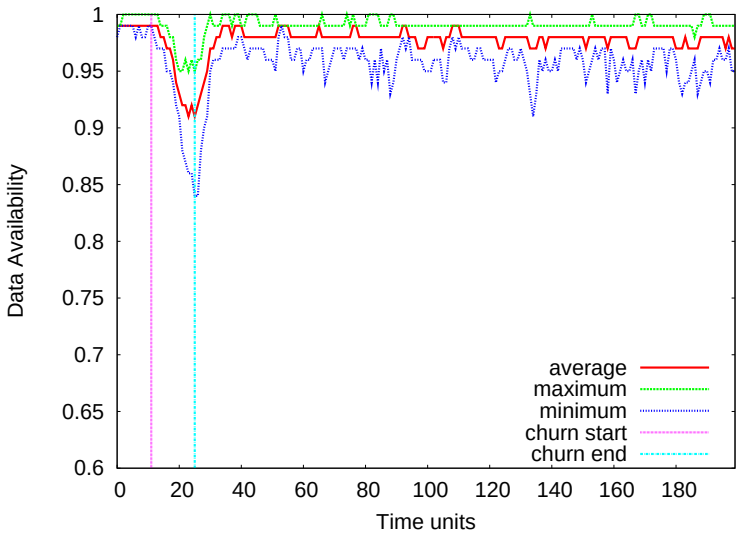


(b) communication costs

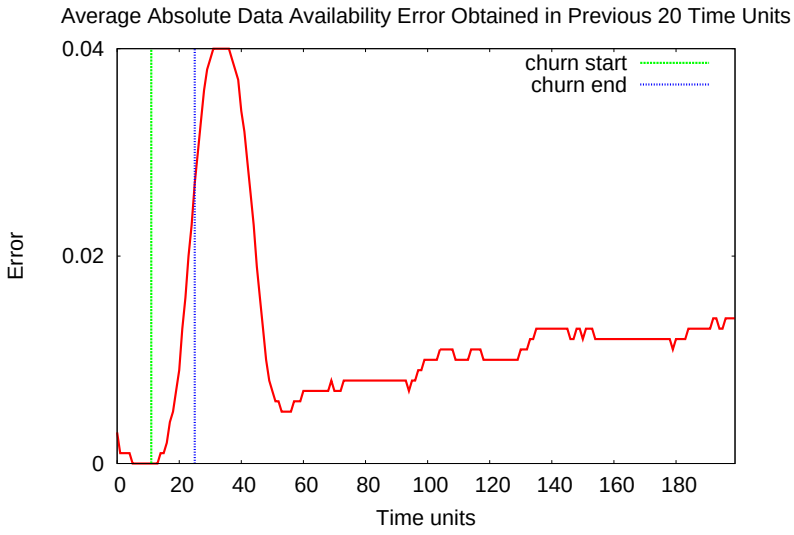
Figure 5.8: Scenario 4: generated costs for low-availability DHT with the average peer availability of 20%. It manages immutable data initially replicated 30 times

over. Again, three curves show the average, maximum, and minimum delivered data availability. Markers "churn start" and "churn end" display when the churn starts, and stops respectively.

The error (Figure 5.9b) goes up during the churn phase, but as early as 20 time units after the



(a) obtained data availability



(b) average error in past 20 time units

Figure 5.9: Scenario 5: highly-available DHT during a churn of a high rate: peer availability drops from 50% to 20% during 15 time units

churn end, the error drops below 0.025. Interestingly, the error peak (0.04) happens out of the churn period (around 30 time units later). Due to the high resolution of Y-axis, it looks that the error increases after time unit 60. However, the average curve on Figure 5.9a demonstrates that

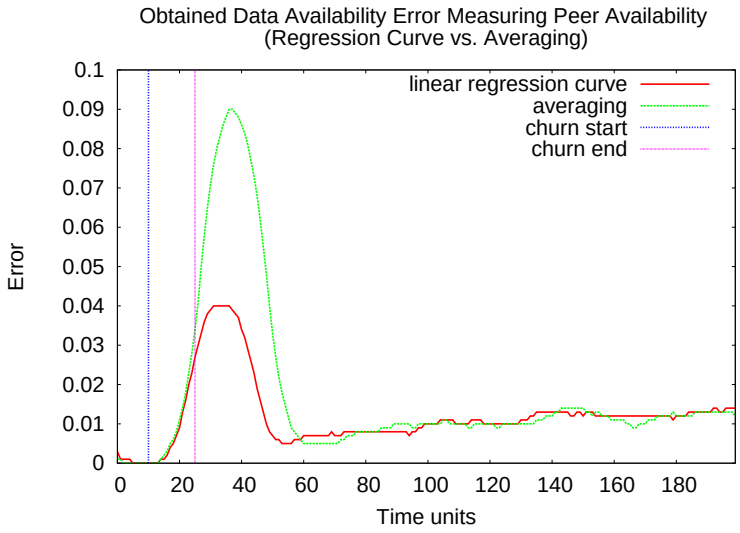


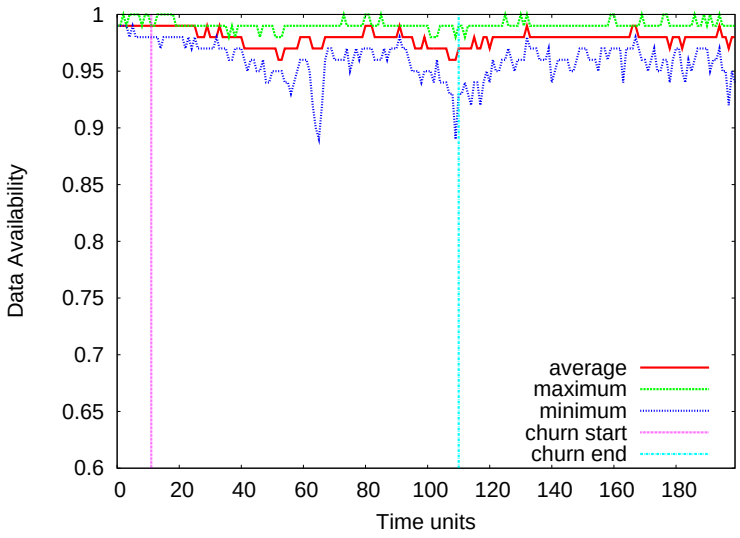
Figure 5.10: The influence of the technique used for measuring peer availability on the obtained data availability error: building the linear regression curve vs. averaging (Section 4.4)

the data availability is pretty stable after recovering from the churn. The curve on Figure 5.9b represents the average error in past 20 time units. Thus, by the definition, it stabilizes slower.

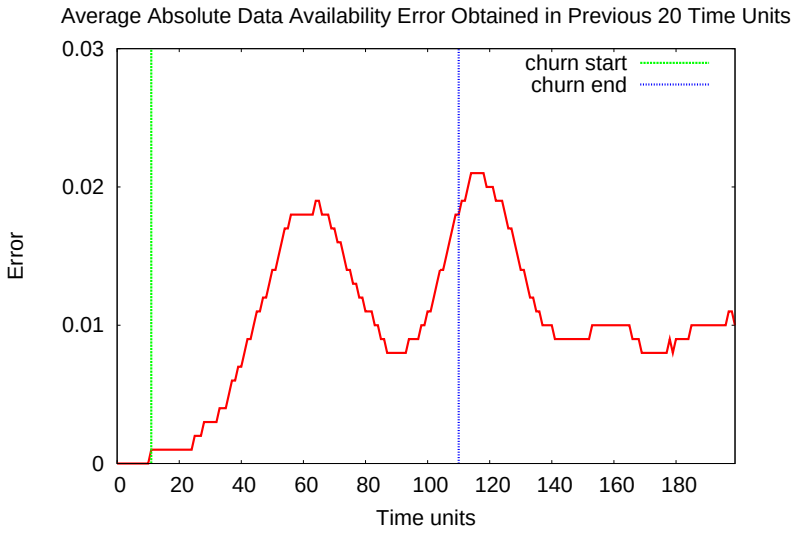
The peer availability measurements rely upon previous measurements stored in local histories. The measured values are higher than the actual ones, because the churn rate is simply too strong to be followed. Consequently, an insufficient number of replicas will be created during churn. Fortunately, the measurement error drops at the end, or shortly after the churn, and peers are able to create enough replicas to recover the data availability.

Recalling the motivation for defining the proposed measurement technique (Section 4.4), we would like to check if it delivers a lower error rate and recovers the data availability faster when a churn happens. Scenario 5 is executed again, but this time we use the protocol that simply averages all received measurements, as initially proposed in Section 4.4. Figure 5.10 compares the error obtained with the one presented in Figure 5.9b. The maximal error is much higher now (0.09 vs. 0.04). Also, availability is recovered a little later (10 time units later). These results confirm our expectations: the proposed measurement technique (Algorithm 8 in Section 4.4) delivers a much lower rate of error during churns and consequently, the guarantees are recovered much faster afterwards.

After a successful recovery from the strong churn, it would be interesting to see what happens if the churn is weaker (Scenario 6). We have used the same DHT as in the previous experiment, but this time the churn rate is much weaker: the peer availability drops from 50% to 20% during 100 time units. Interestingly, our protocol has coped very well with the churn, and the requested



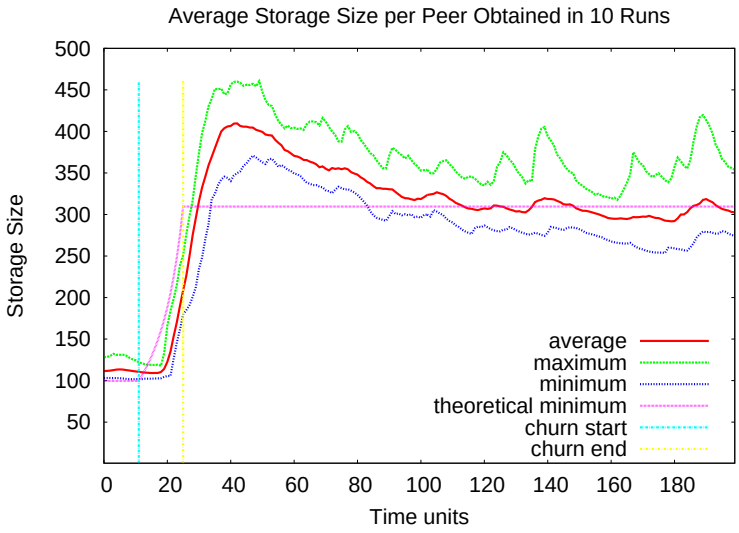
(a) obtained data availability



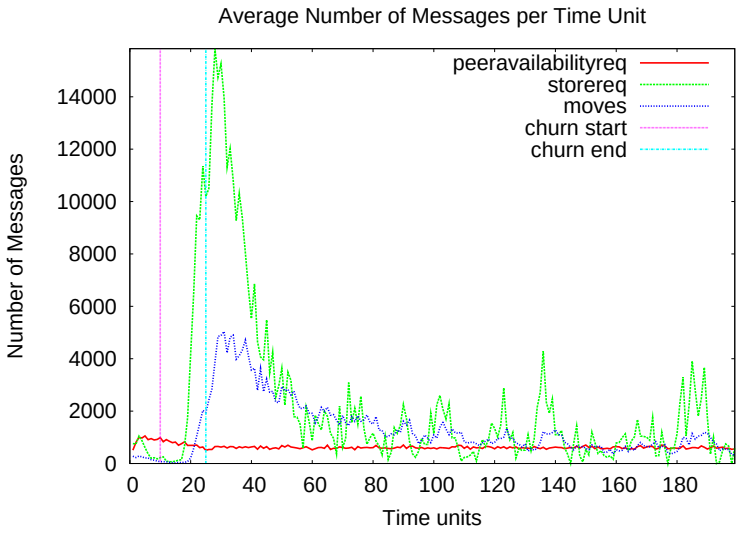
(b) average error in past 20 time units

Figure 5.11: Scenario 6: highly-available DHT during a churn of a weak rate: the peer availability drops from 50% to 20% during 100 time units

data availability is practically maintained (Figure 5.11) throughout the churn phase. The error (Figure 5.11b) reaches its maximum of 0.02 at the end of the churn phase. The DHT handles this kind of churn well thanks to the applied measurement method (Section 4.4). During long



(a) average storage size per peer



(b) communication costs

Figure 5.12: Scenario 5: generated costs for highly-available DHT during a churn of a strong rate: peer availability drops from 50% to 20% during 15 time units

churns, regression curves built have a very precise slope, and measured peer availability is close to the real one. Thus, peers are able to react properly, and follow the churn.

Figure 5.12 presents the costs generated in the DHT equipped with our protocol given a strong

churn. As one can see, already during the churn, the peers start the data availability recovery by creating additional replicas (Figure 5.12a), until the required availability is reached. After that, the storage costs remain close to the theoretical minimum. The costs generated during recovery from the weak churn (Figure 5.13a) follow the same pattern.

As demonstrated in Section 5.5.1, the number of *StoreRequest* messages is correlated with increasing the average storage overhead. When a churn occurs, the protocol needs to recover the previous data availability, and therefore, the number of messages goes up. If the churn is very strong (Figure 5.12b), the peak of the "storereq" curve happens when the churn is over, and the peer availability has already stabilized. This is because the number of online peers has decreased drastically in a short period. Suddenly, the DHT does not contain enough replicas of managed objects, and this is the situation that we already had in Scenarios 1 and 2. Thus, a part of the graph after "churn end" and the curves in Figure 5.3b and 5.4b are similar, and the discussion about the number of *StoreRequest* and "moves" messages still applies.

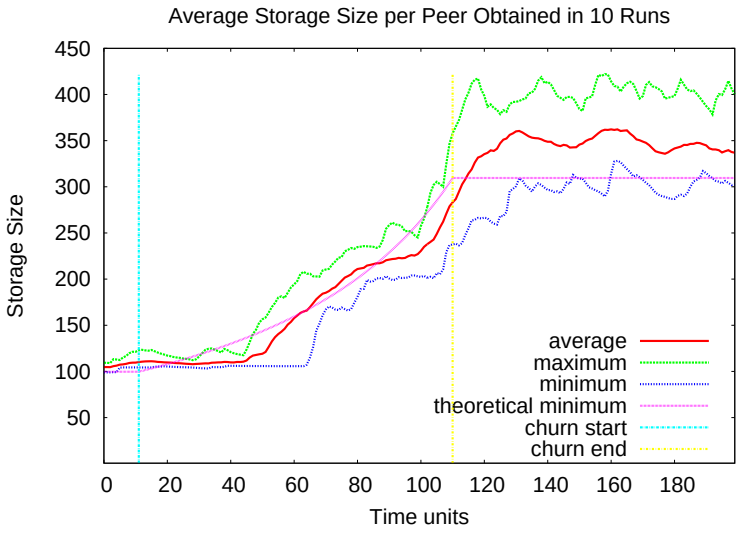
During and after the strong churn only a smaller number of peers comes online and adjusts the number of replicas locally. However, recovering the average data availability requires that more peers become available, measure the new peer availability and react appropriately. Such a process is much longer than the duration of the strong churn. In contrast, if the churn is slower (Figure 5.13b), many peers have the chance to come online during the churn, and the guarantees can be maintained.

As in the scenarios without churns (Scenarios 1-4), the cost of measuring current peer availability does not depend on system properties such the actual peer availability, or DHT size. It depends only on the measurement precision we want to achieve. As defined in Section 5.1, peers measure the average peer availability immediately after coming online. Therefore, the total number of *PeerAvailabilityRequest* messages per time unit is related to the number of peers that do measurements. The "peeravailabilityreq" curve on Figure 5.12b and 5.13b demonstrate this statement: the measurement costs are quite independent from the initial settings, and the obtained data availability. It depends only on the peer online arrival rate. Before the churn, the arrival rate is higher than afterwards, and thus the number of *PeerAvailabilityRequest* messages is lower after the churn.

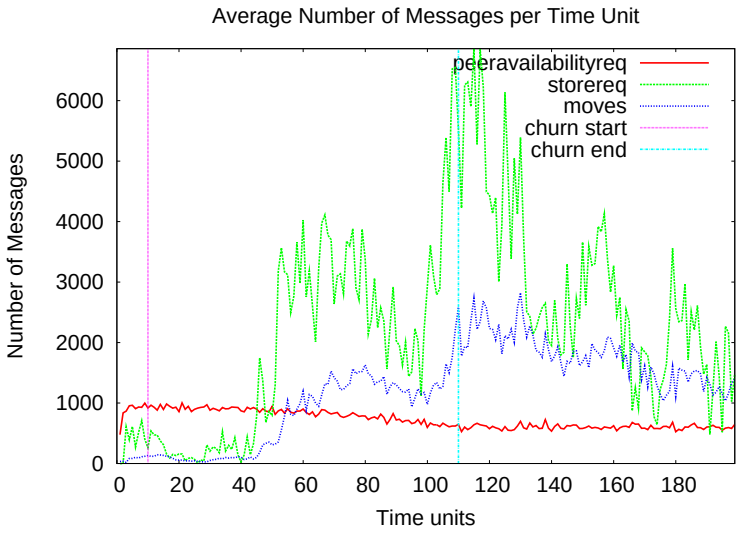
Maintaining Data Availability

As expected, a "negative" churn of any kind (Scenario 7 and 8) does not have any effect on the availability of data provided. However, if peer availability increases, fewer replicas are needed to maintain the same data availability, and, as in Scenario 3 and 5, storage costs could be reduced.

Figure 5.14a and Figure 5.15a demonstrate the evolution of the storage costs in DHTs where the average peer availability increases over time. Our protocol is able to detect the new stable situation, and to reduce the number of replicas accordingly. At the same time, the number of measurement messages remains stable (Figure 5.14b and Figure 5.15b). The number of replicas moved is insignificant, because increasing peer availability stabilizes the DHT routing rules.



(a) average storage size per peer

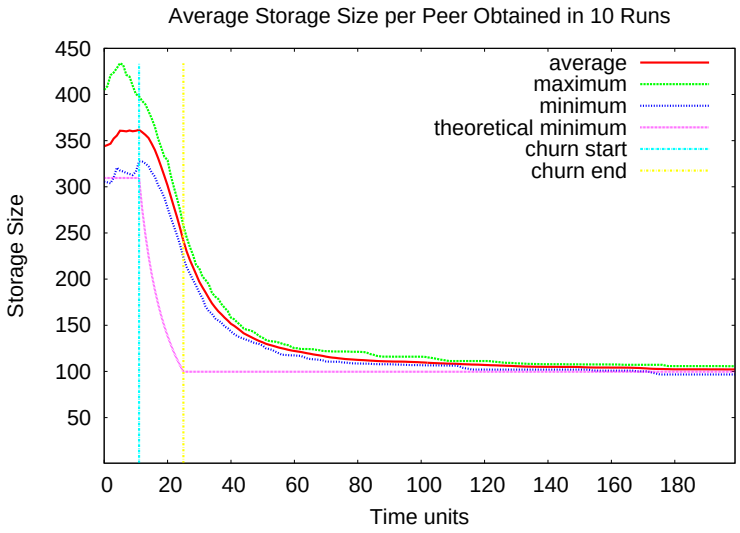


(b) communication costs

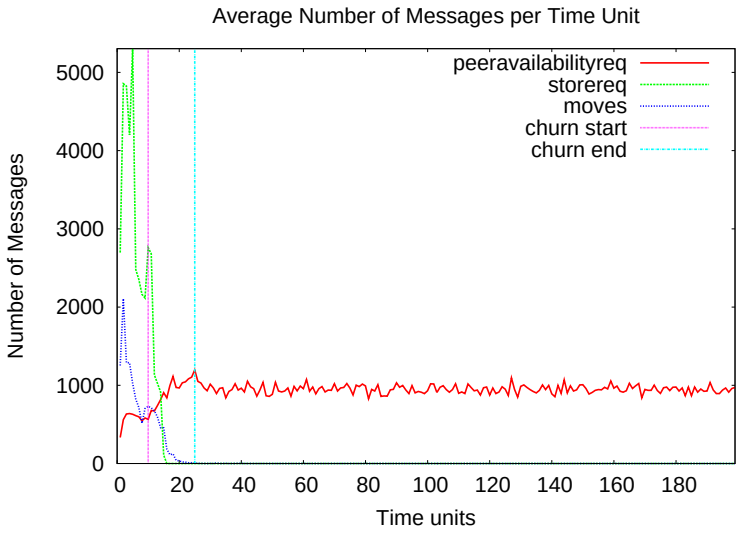
Figure 5.13: Scenario 6: generated costs for highly-available DHT under a churn of a strong rate: the peer availability drops from 50% to 20% during 100 time units

Conclusion

This Section has tested if the proposed protocol can handle churns that appear very often in reality. The results presented show that data availability can be recovered to the level it was being



(a) average storage size per peer

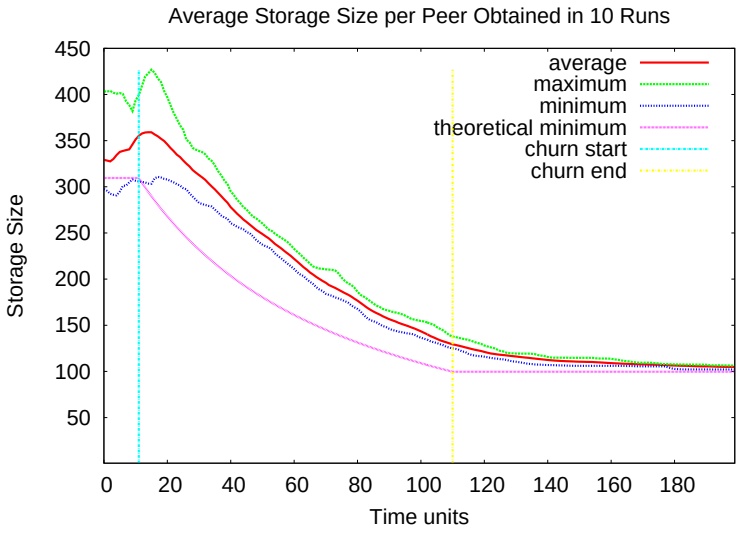


(b) communication costs

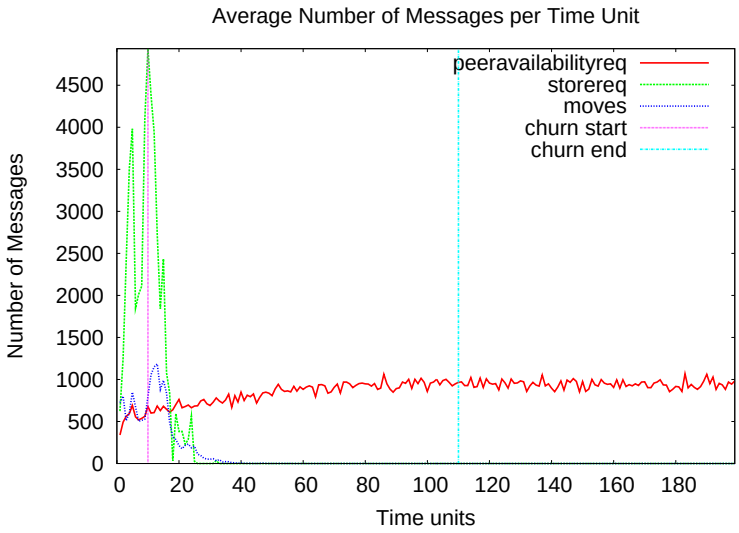
Figure 5.14: Scenario 7: generated costs for low-availability DHT under a "negative" churn of a strong rate: the peer availability increases from 20% to 50% during 15 time units

delivered at before after the strong churn is over. If the churn rate is weaker, data availability is practically maintained throughout the whole churn period.

Also, measuring peer availability by building regression curves is justified. It is compared



(a) average storage size per peer



(b) communication costs

Figure 5.15: Scenario 8: generated costs for low-availability DHT during a "negative" churn of a weak rate: the peer availability increases from 20% to 50% during 100 time units

with a protocol that calculates the peer availability just by averaging collected measurements. The results obtained confirm that the technique delivers a far lower error rate during churns and consequently makes recovery of guarantees much faster.

Finally, we have observed how our protocol can deal with a "negative" churn of an arbitrary rate. It does not have any influence on the data availability. However, if peer availability remains stable some time after the churn, the number of replicas is reduced,thus freeing space in peer's storages.

5.6 Managing Guarantees of Mutable Data

A facility for updating already stored objects is essential for many applications. Therefore, the same set of scenarios is executed again, but this time allowing updates.

5.6.1 Stable Peer Availability

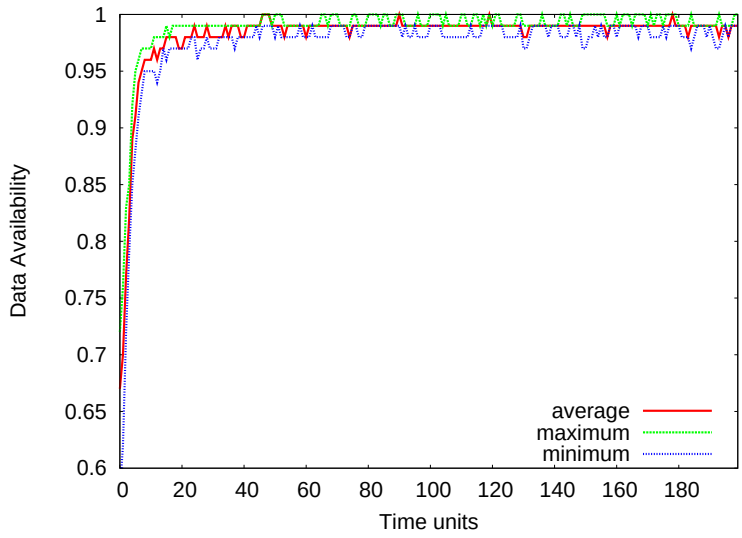
Reaching Data Availability

Scenario 1 and 2 are executed again, allowing that the managed data can be modified with the given update rate. As in the previous Section, the DHT equipped with our protocol should detect that the requested data availability is not delivered, and create more replicas of managed objects. At the same time, access to updated objects must return the latest value with the probability of 99%.

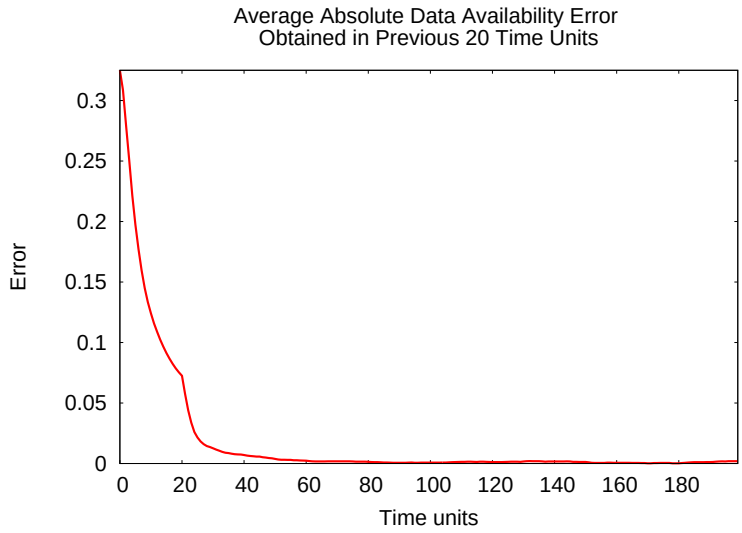
Figure 5.16 and 5.17 show the results achieved. The required data availability is reached, while keeping consistency at the requested level with an insignificant error rate. In order to see if updates produce any additional effect, we compare Figure 5.16b with Figure 5.1b and Figure 5.17b with Figure 5.2b. The time needed for delivering the guarantees is much the same, i.e. allowing updates does not influence it. However, our protocol maintains data availability closer to the availability requested, if data are modified during the simulation. This is a positive side-effect of a high update rate. Modifying an object requires updating all its available replicas, and inserting the new version of those currently offline. As we have seen in Section 4.6, the total number of replicas is higher than R enabling the delivered data availability to be even closer to the requested one.

By comparing the cost side (Figure 5.18a and 5.19a) with the cost of managing immutable data (Figure 5.3a and 5.4a), one can notice similar behavior. The increase of storage overhead follows the same pattern: it increases significantly at the beginning, but soon after reaching the requested data availability, its size stabilizes. Managing mutable data produces a higher overhead, but still much lower than the theoretical maximum defined by Formula 4.38. Obsolete replicas are eventually removed, either when the topology changes, or during peer availability measurements. As Figure 5.18a and 5.19a show, the actual costs are approximately 2.6 and 1.6 times higher than the predicted minimum defined by Formula 4.33.

The number of replicas of an object increases with the number of updates performed, but its maximum is bounded to $\frac{R}{p}$ (see Section 4.6.2). Thus, the average storage size depends on the actual objects' update probability. The higher it is, the closer average storage size is to the defined maximum. To confirm that, Scenario 1 is executed additionally with an update



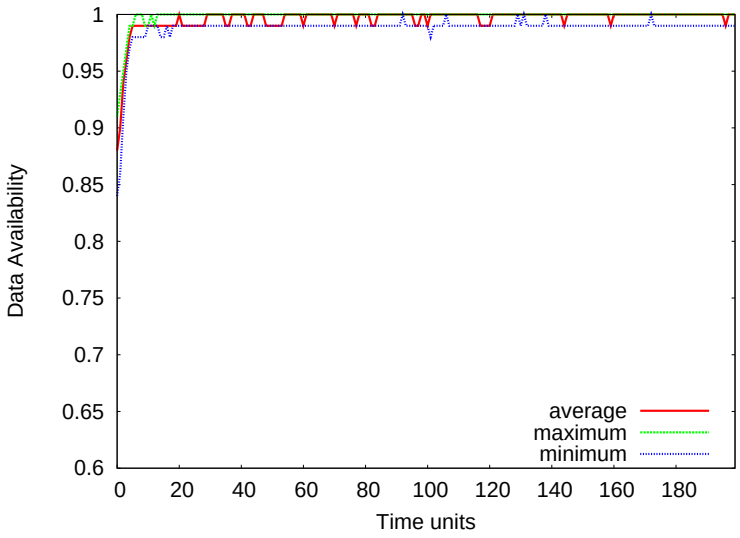
(a) obtained data availability



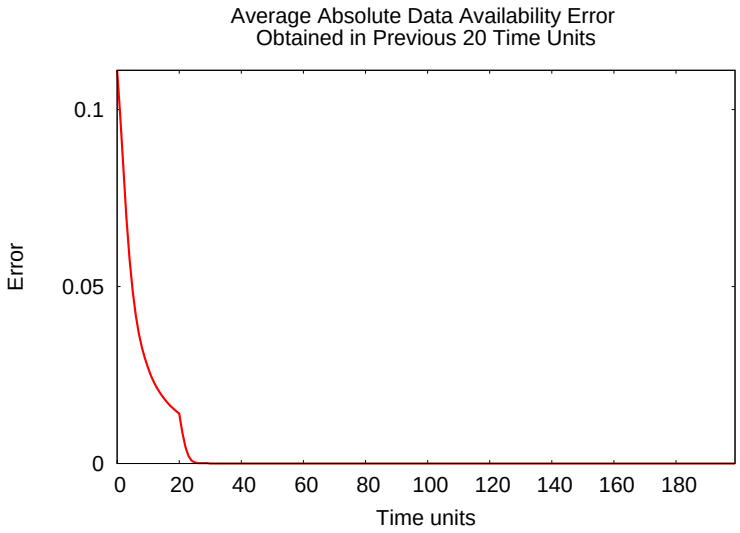
(b) average error in past 20 time units

Figure 5.16: Scenario 1: low-availability DHT with peer availability of 20%. Mutable data are replicated initially 5 times, but at least 21 replicas are needed to ensure 99% data availability. The update distribution is uniform with a probability of 10% (Scenario 1)

probability of 20% and 5%. The average storage size increases to 1100 replicas per peer for the higher update rate. Reducing the update probability to 5% makes peers keep only about 700



(a) obtained data availability

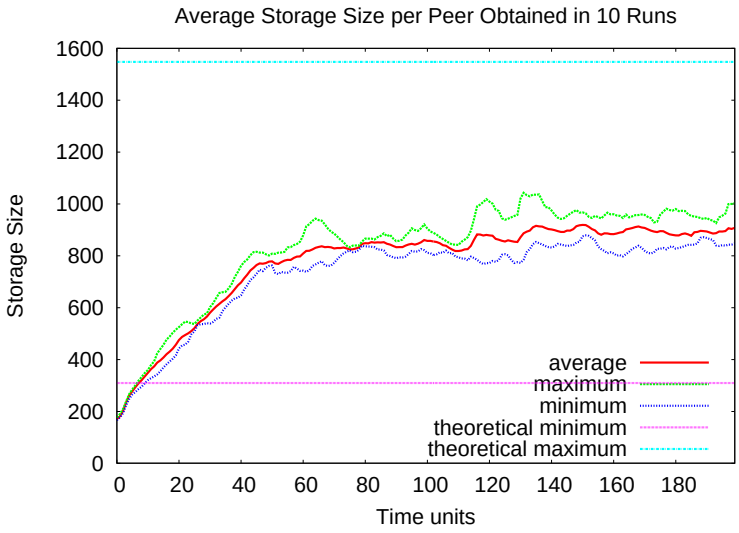


(b) average error in past 20 time units

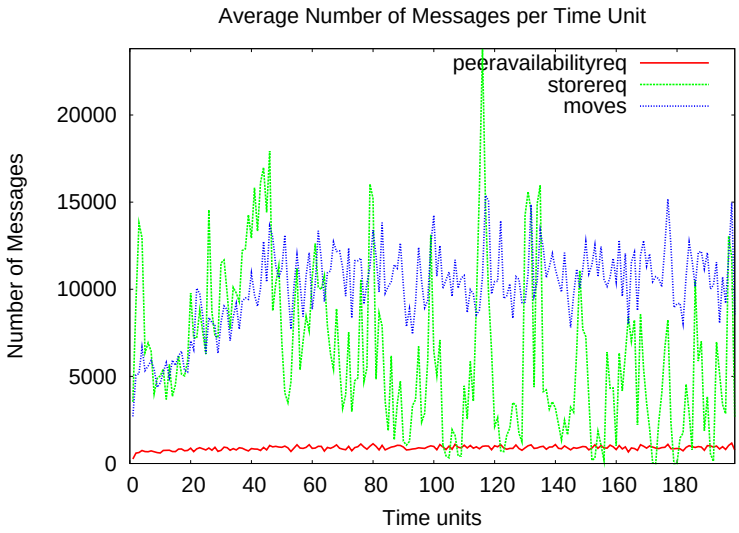
Figure 5.17: Scenario 2: highly-available DHT with peer availability of 50%. Mutable data are replicated initially 3 times, but at least 7 replicas are needed to ensure 99% data availability. The update distribution is uniform with a probability of 10% (Scenario 2)

replicas in their storages.

Measuring peer availability (the "peeravailabilityreq" curve in Figure 5.18b and 5.19b) gen-



(a) average storage size per peer

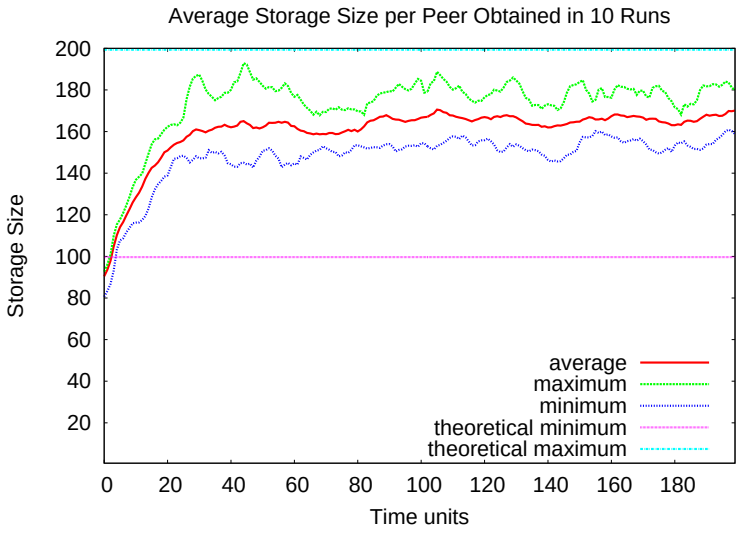


(b) communication costs

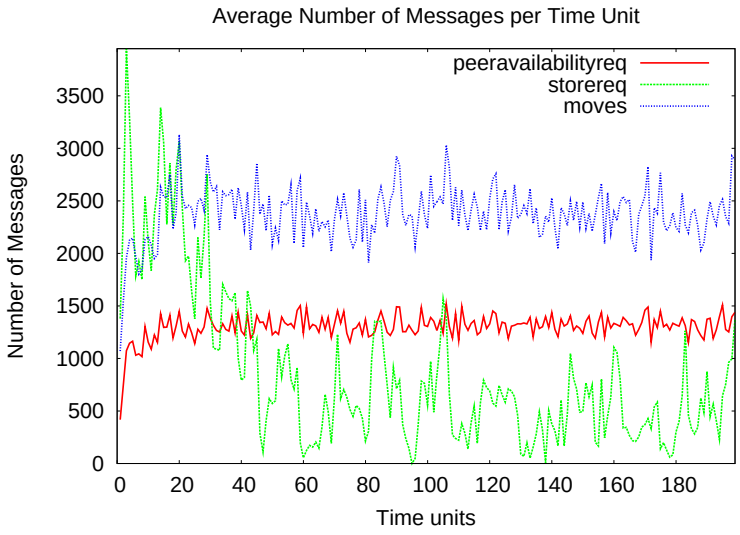
Figure 5.18: Scenario 1: generated costs for low-availability DHT with the peer availability of 20%. Mutable data are replicated initially 5 times

erates the same volume of messages as before (Figure 5.3b and 5.4b), i.e. it is not influenced by updates.

More replicas are moved around (the "moves" curve) than when managing immutable data.



(a) average storage size per peer



(b) communication costs

Figure 5.19: Scenario 2: generated costs for highly-available DHT with peer availability of 50%. Mutable data are replicated initially 3 times

This is due to a higher number of replicas managed by the system. The new replicas are created for two reasons: peers want to achieve the requested data availability, and objects are updated frequently. With every topology change, some old version must receive missed updates. As

described in Section 4.3, this is realized by moving the new version towards the old one. Since the update rate is fairly high, peers are constantly moving replicas towards their old version. Someone could say that this traffic is too high, but the number of moved replicas represents only about 3% of all managed replicas in the DHT.

To confirm that the number of moved replicas depends on the applied update rate and stability of the underlying DHT, Scenario 1 is executed, varying the update rate and the online session time. Figure 5.20a demonstrates that the number of moved replicas increases with an increasing update rate. On other hand, if a peer's sessions are longer, the DHT topology becomes more stable, and therefore the number of moved replicas decreases.

As one can see, the number of *StoreRequest* messages goes up during the period of reaching the guarantees, but afterwards it does not disappear fully as is the case when data are not modified at all (compare to Figure 5.3a and 5.4a). The traffic is especially visible in the low-availability DHT (Scenario 1). There are two reasons for this. The storage overhead is 2.6 times higher than before. If the measured peer availability is lower than the actual one, a peer could wrongly decide (especially when the peer availability is low, see Formula 4.5) to create additional replicas of all objects locally owned. In combination with the higher storage overhead, the number of *StoreRequest* messages becomes higher too. Second, it seems that a high update rate influences the precision of the defined measurement mechanism, making the error rate higher. To confirm or reject this doubt, we have calculated the cumulative probability distribution of the achieved error by measuring peer availability under different update rates. Figure 5.21 shows that the update rate does have an influence on the obtained error. As we can see, if the update rate is low (5%), the error is almost the same as when data are immutable. However, by increasing the rate, the chances of having a higher error rate increase.

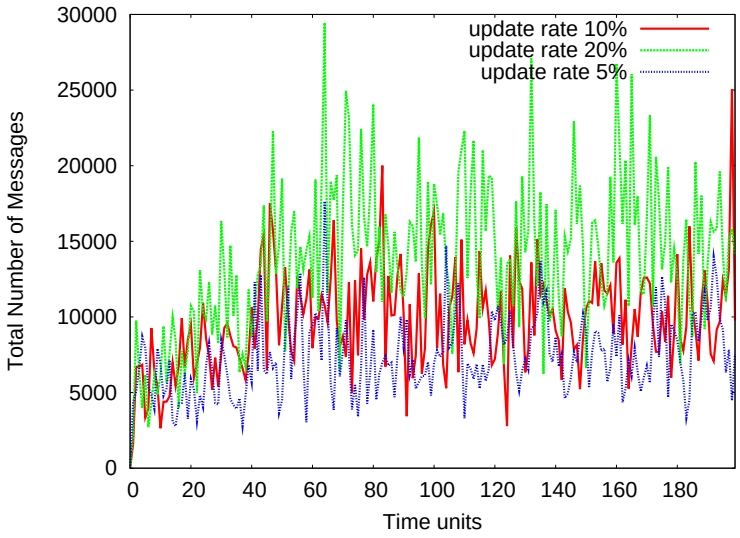
Thus, we can conclude that the number of *StoreRequest* is higher when mutable data are managed only due to the the update rate (10%)applied, because the storage overhead also depends on it.

Managing Data Availability

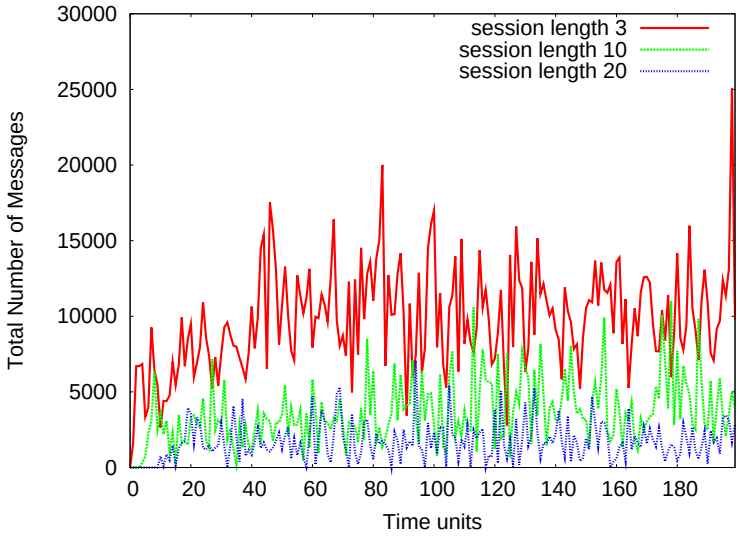
It would be interesting to see how our protocol performs when the initial number of replicas is higher than needed (Scenario 3). As shown in Section 5.5.1, our protocol has been able to manage the availability of immutable data while reducing the number of replicas to the minimum needed.

If we manage mutable data, data availability and consistency are preserved, but the storage costs are somewhat different (Figure 5.22a). Due to the high update rate they cannot be close to the minimum any more, but remain fairly stable and on the same level as in Figure 5.19a.

Since Scenario 2 and 3 are based on the same DHT, the number of moved messages and the costs of measuring peer availability are at the same level in both cases (see the curves "moves" and "peeravailability" on Figure 5.22b and 5.19b). Data availability and consistency are delivered from the beginning in Scenario 3. Thus, the number of *StoreRequest* messages is much lower than in Scenario 2. As before, the later activities are related to the error in measuring peer



(a) varying update rate



(b) varying session length

Figure 5.20: The number of moved replicas (Scenario 1) as a function of the update rate and the online session length

availability.

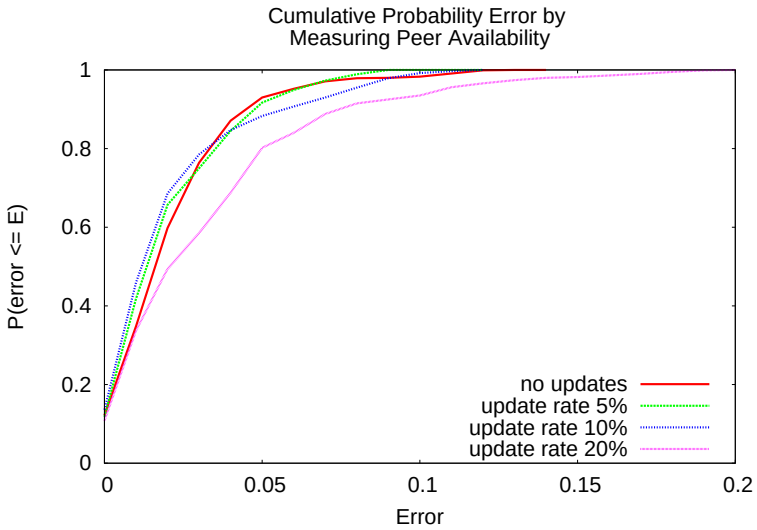


Figure 5.21: The cumulative probability distribution of the obtained error rate by measuring peer availability in Scenario 1 as a function of different update rates: no updates, 5%, 10%, and 20%

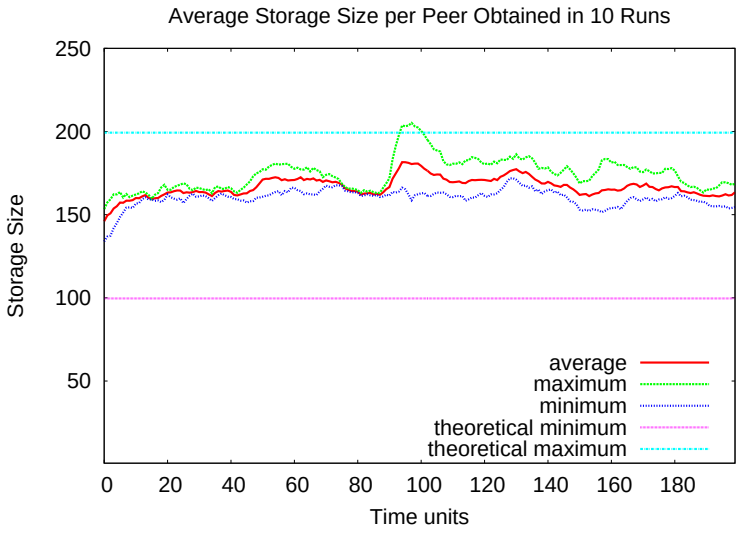
Conclusion

This Section has shown that the protocol presented in this thesis is able to reach and maintain the requested availability and consistency levels even if the managed data are mutable. Moreover, it demonstrates that this is feasible under a high update rate - usually much higher than a real one. The price has been payed on the cost side. The storage overhead is higher than before, but it depends on the update rate applied . Even with an update rate of 10%, it is far away from the predicted theoretical maximum. Updates increase the number of replicas being managed in the DHT. Thus, a change of DHT topology makes more replicas candidates for moving. Due to a measurement error, a peer could decide to create additional replicas of all objects managed in its local store. Since the number of locally stored replicas is higher, the generated traffic will be higher too. Finally, as one might expected, the cost of measuring peer availability does not depend on the nature of managed data; it is on the same level as when updates were not allowed.

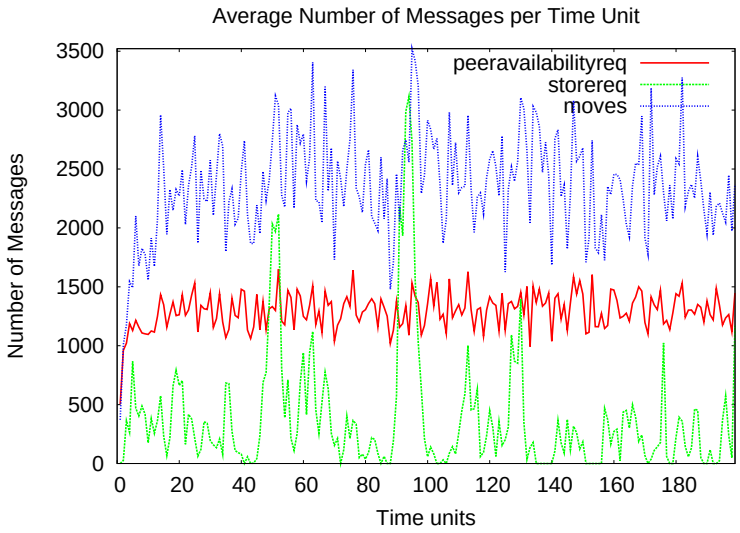
5.6.2 DHTs Under Churn

Recovering Data Availability

The previous tests have confirmed that a DHT equipped with our replication protocol can reach the requested data availability and consistency, while managing mutable data under a high update



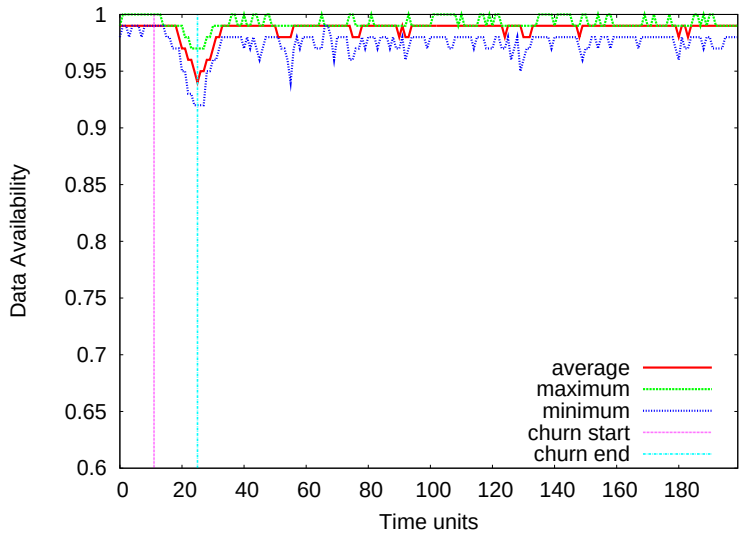
(a) average storage size per peer



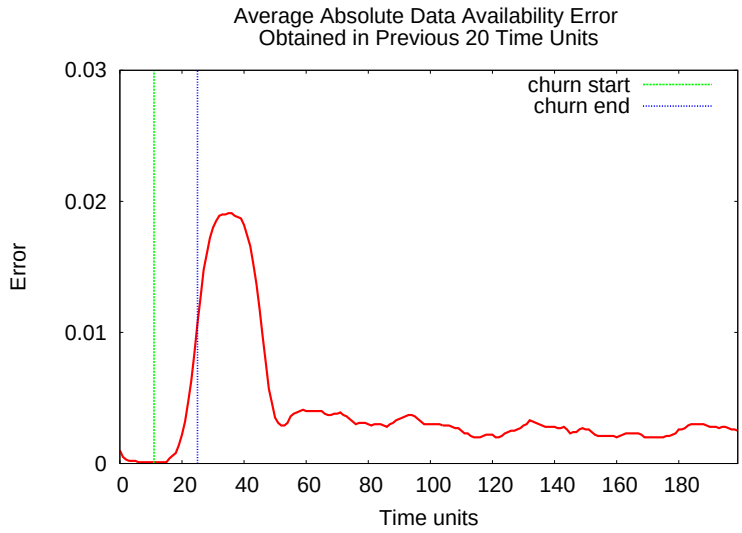
(b) communication costs

Figure 5.22: Scenario 3: generated costs for highly-available DHT with the peer availability of 50%. Mutable data are replicated initially 11 times (Scenario 3)

rate. The following simulations go a step further; they test if the requested data availability and consistency can be recovered after or maintained during a churn, while managing mutable data. The update distribution remains the same as in the previous Section.



(a) obtained data availability



(b) average error in past 20 time units

Figure 5.23: Scenario 5: highly-available DHT during a churn of a strong rate: the peer availability drops from 50% to 20% during 15 time units. Mutable data are subject to updates according to the uniform distribution of 10%

The protocol is first tested during a strong churn (Scenario 5). The error (Figure 5.23b) reaches its maximum (0.02) approximately 5 time units after the churn end. Compared to the case when

updates are not allowed (Figure 5.9b), the obtained error is lower. This is due to the high update rate. Every update creates more replicas, increasing the chances of finding at least one of them online.

The evolution of the average storage size is as expected (Figure 5.24a). It goes up until the requested levels of data availability and consistency are reached. Afterwards, it remains stable and on the same level as in Figure 5.19a. The number of generated *StoreRequest* messages (Figure 5.24b) follows the pattern seen already. During the recovery phase, it increases significantly, because additional replicas need to be created. After reaching the guarantees, it decreases to the same level as in Scenario 2 and 3. The reasons for such a behavior are given in the previous Section. The other two curves are much the same as those in Figure 5.12a. Only the number of moved replicas is higher due to the high update rate.

As one might expected, managing mutable data during a weaker churn produces better results. The average error (Figure 5.25b) is not higher than 0.005, and most of the time much lower. Thus, the requested data availability and consistency is maintained practically throughout the whole simulation. These are better results than those for managing immutable data (Figure 5.11b). Again, updating objects increases their availability.

The generated costs (Figure 5.26a) follow the previously discussed patterns.

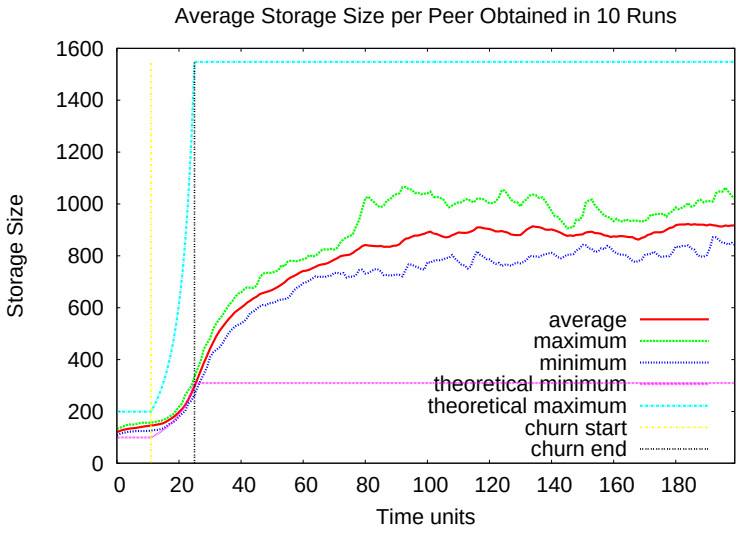
Managing Data Availability

Scenarios 7 and 8 explore the possibility of having "negative churns" in a DHT, i.e. periods when the average peer availability increases. Such a situation does not affect the data availability and consistency guarantees, since the number of replicas were computed for a lower peer availability. However, if the new peer availability remains stable over a longer period of time, there is no need to keep so many replicas in the DHT; the same data availability is achieved even with fewer replicas. Figure 5.27a shows the change of the storage costs in the DHT equipped with our protocol. It is able to detect the new stable situation, and to reduce the number of replicas accordingly. The results for Scenario 8 (Figure 5.28a) are similar.

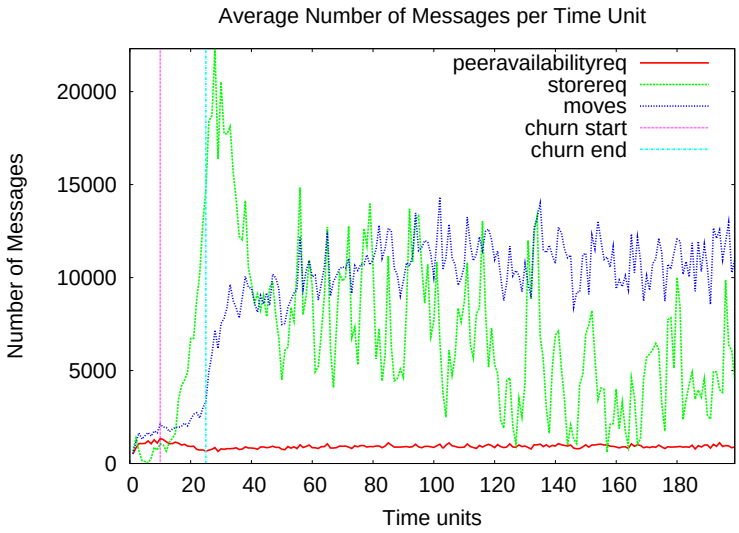
Removing unnecessary replicas also decreases the number of replicas that are moved due to the DHT topology changes (Figure 5.27b and 5.28b). The measurement costs do not differ from those in Figure 5.14b. The "storereq" curve shows the activity due to errors in measurement. At the beginning, the peer availability is low, and the system contains much more replicas (304000) than after the churn (64000). Therefore, any measurement error produces much higher traffic before the churn.

Conclusion

This Section has tested if our protocol can recover or maintain the requested data availability and consistency while allowing updates. Depending on the update rate, the guarantees are delivered with at least the same error rate as when data are immutable. A higher update rate can only make the error rate smaller. On other hand, modifying object more frequently increases the



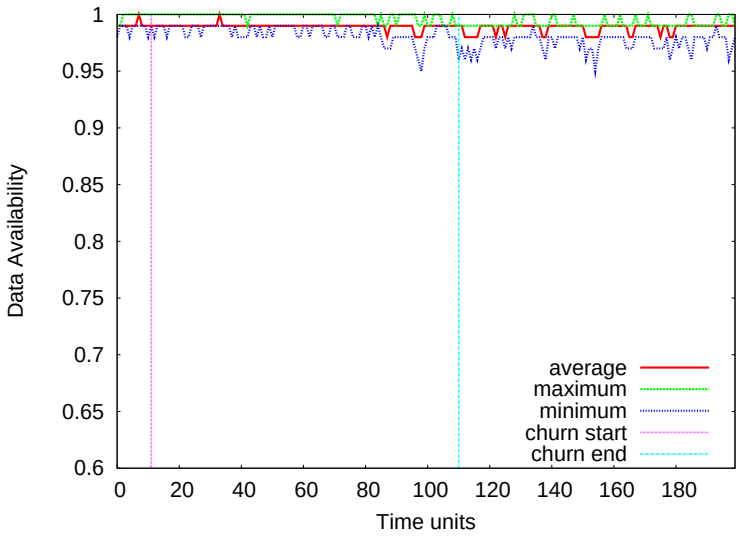
(a) average storage size per peer



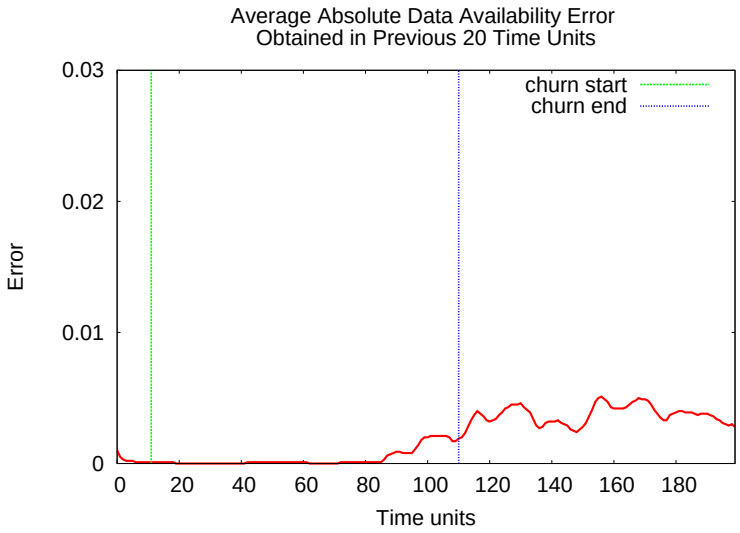
(b) communication costs

Figure 5.24: Scenario 5: generated costs for highly-available DHT under a churn of a strong rate: the peer availability drops from 50% to 20% during 15 time units. Mutable data are subject to updates according to the the uniform distribution of 10%.

storage overhead and the traffic related to moving replicas due to topology changes and creating additional replicas.

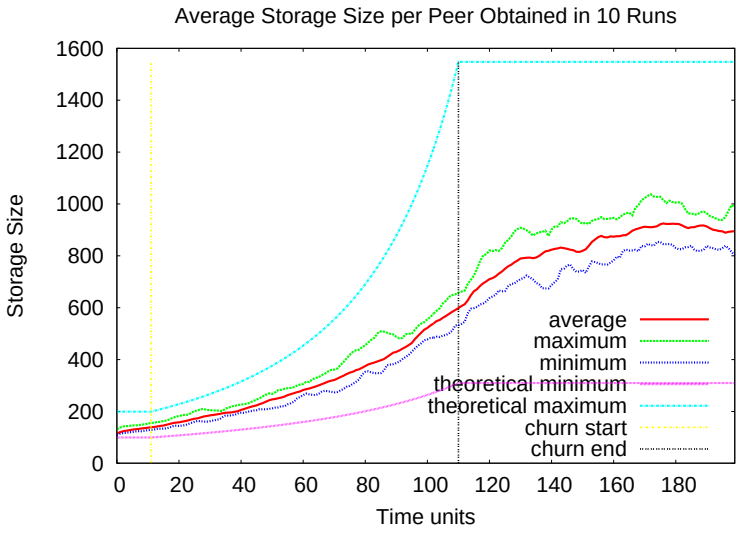


(a) obtained data availability

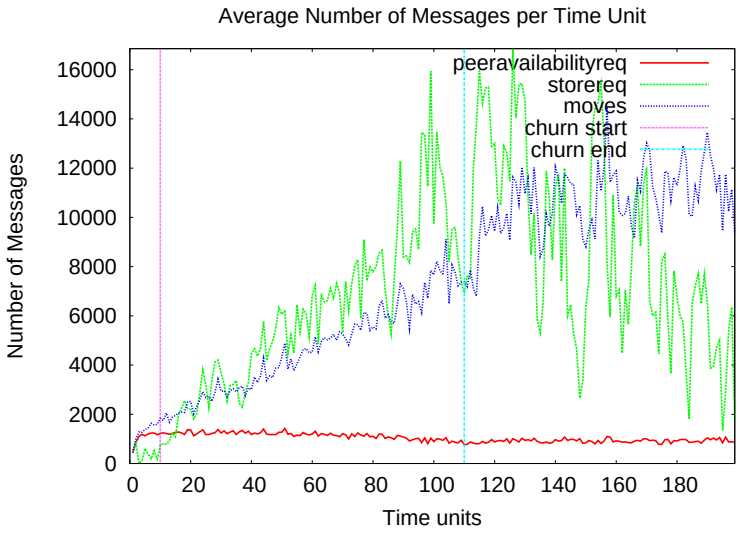


(b) average error in past 20 time units

Figure 5.25: Scenario 6: highly-available DHT during a churn of a weak rate: peer availability drops from 50% to 20% during 100 time units. Mutable data are subject to updates according to the uniform distribution of 10%



(a) average storage size per peer

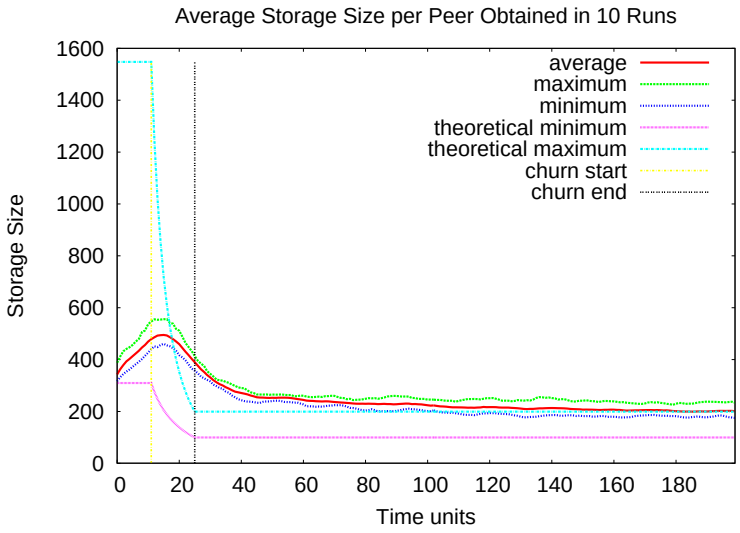


(b) communication costs

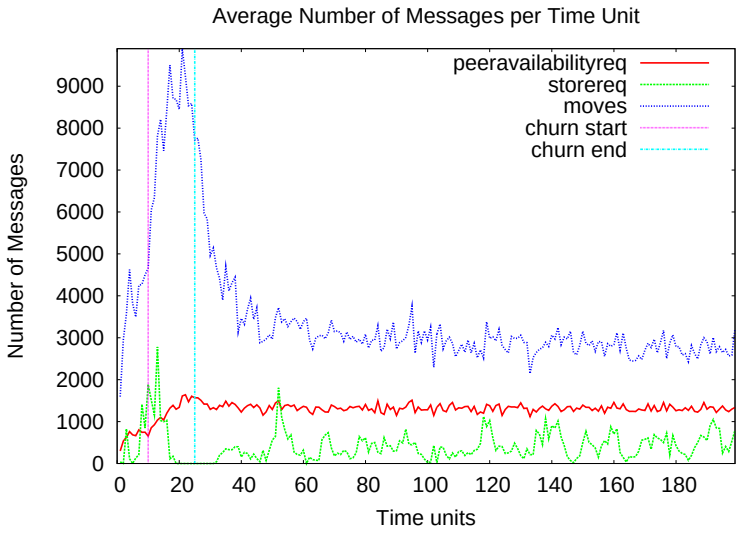
Figure 5.26: Scenario 6: generated costs for highly-available DHT during a churn of a strong rate: peer availability drops from 50% to 20% during 100 time units. Mutable data are subject to updates according to the uniform distribution of 10%

5.7 Summary

This chapter has presented the results of the experiments that aimed to test whether our protocol can deliver the high availability and consistency of data managed in a highly dynamic DHT with



(a) average storage size per peer

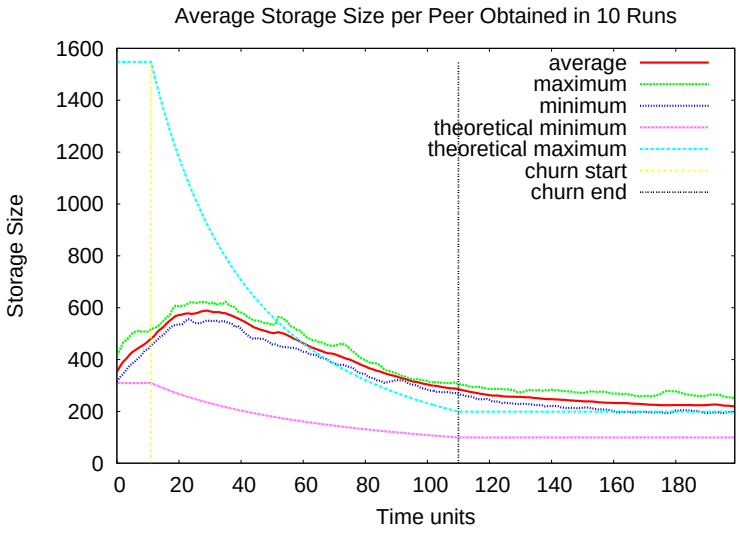


(b) communication costs

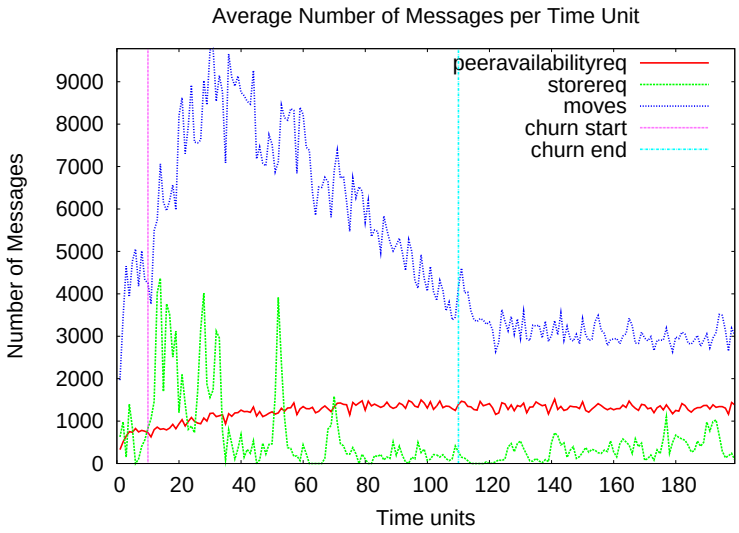
Figure 5.27: Scenario 7: generated costs for low-availability DHT under a churn of a strong rate: the peer availability increases from 20% to 50% during 15 time units. Mutable data are subject to updates according to the uniform distribution of 10%.

arbitrary and previously unknown peer availability.

For this purpose, a simulator was written. It allowed us to test different important scenarios in



(a) average storage size per peer



(b) communication costs

Figure 5.28: Scenario 8: generated costs for low-availability DHT under a churn of a strong rate: the peer availability increases from 20% to 50% during 100 time units. Mutable data are subject to updates according to the uniform distribution of 10%

a convenient way. A great deal of system information has been collected into log files used later for plotting various graphs. The results obtained can be summarized as follows:

- A DHT equipped with the proposed protocol is able to reach, maintain and recover the high availability and consistency of both mutable and immutable data
- Guarantees are delivered without knowing peer availability in advance and independently of the initial replication factor, and the actual update rate
- Updates reduce the error between the achieved and requested guarantees
- The requested data availability and consistency are maintained during weak churn with an acceptable error
- The storage overhead introduced matches the predicted one
- Updates increase the storage overhead. However, even with a high update rate of 10%, it is around 2.6 times higher compared to the case when immutable data are managed in a low-availability DHT
- The costs related to measurement of peer availability depend on the requested precision only
- Higher update rates ($> 5\%$) decrease the precision of the measured peer availability
- An error in measuring peer availability could generate unneeded requests to add new replicas. If data are updated with a higher rate, such unwanted traffic is increased additionally
- The number of replicas that are subject to being moved due to a topology change depends on the update rate and the online session duration

Chapter 6

Conclusion

THIS thesis has demonstrated how high availability and consistency of data can be achieved and maintained in a dynamic DHT. The replication protocol designed is informed during the deployment phase of the data availability and consistency guarantees that should be delivered at run-time. At least once per online session, every peer measures the actual peer availability in the DHT, and based on the obtained value the replication factor of the locally managed data will be potentially adjusted in order to meet the requested guarantees.

The solution presented is fully decentralized and requires no advanced knowledge about the underlying DHT in advance. It adapts to an arbitrary peer availability and recovers the requested guarantees shortly after churns. If the churn rate is low, it maintains them all the time. Clearly, deploying a DHT equipped with our protocol brings advantages in environments whose nature is unknown or hard to predict. The protocol ensures that high data availability and consistency will be achieved without the need for external assistance or re-configuration.

6.1 Achievements

Every replication protocol needs to manage mappings between replicas and their locations in the system. This is provided by a replica directory, usually placed on a reliable, dedicated machine. DHTs deployed in practice do not allow such a configuration, because all participating peers are fully autonomous and unreliable. Therefore, the first step towards defining the protocol presented in this thesis has been designing the decentralized replica directory. It is built on top of the DHT, where replica location is equal to a key under which the replica is inserted into the system. Hence, instead of keeping the mappings in a central place, they are managed by using a key generation function known by every peer. It correlates an object key with all its replica keys, i.e. by knowing an object key (provided by the application), we are able to compute all replica keys just with the local knowledge. Afterwards, replicas could be accessed according to some logic. As the evaluation shows, the defined key generation mechanism ensures that a

peer manages only one replica of an object with a very high probability. Otherwise, the requested availability and consistency could never be guaranteed, because determining the right replication factor is based on this assumption.

We have focused on guaranteeing a high availability of data managed within the DHT assuming that they are not modified at all after insertion. To achieve this transparently, data are stored a number of times in the DHT. Later on, when they are about to be accessed, it is enough to find any of the replicas. The analysis performed has shown that determining the right number of replicas depends on the pre-configured data availability and the actual peer availability. Since the peer availability can and does change over time, maintaining data availability at the requested level requires measuring the actual availability of peers and adjusting the replication factor of managed data accordingly.

Afterwards, we relaxed the settings by allowing management of mutable data. Hence the protocol has been extended to additionally deliver high consistency of modified data. The consistency model designed belongs to the group of weak ones, but unlike the others, it provides arbitrarily high probabilistic guarantees on the consistency of available data until all replicas are eventually synchronized and data are fully consistent. Besides updating all available replicas at that moment, achieving such guarantees is only possible if the new version of offline replicas is inserted into the DHT as well. The obsolete replicas are synchronized by using the push approach: the peer that has the new version informs the peer that manages an old version about missing updates. Since data are potentially mutable now, finding just any of the available replicas is not sufficient when data are requested. Instead, the protocol returns the replica with the highest available version number. The number of replicas that should be modified/inserted does not differ from the case when immutable data are managed. On the other hand, the total number of replicas of an object is higher now. It depends on the applied update rate, but has an upper bound.

The existence of the version number helps to coordinate concurrent updates. An update increments the version number of the previously accessed object. Also, replicas are addressed in a predefined sequence. Therefore, the version number will be the same across all concurrent updates on an object. The proposed coordination allows only one update to get through. The rest of them fail and must be compensated by the application. Hence, a replica update will fail if it already has the same version number.

A very important outcome of the previous analysis shows that this replication protocol can be self-adaptable to environment settings only if it is able to measure the actual peer availability and adjust the number of replicas of managed data according to the formulas derived. We are unable to probe a peer directly, because the protocol can access the DHT only by using the standard DHT functions that are not aware of the underlying topology. Thus, peer availability is measured indirectly via replica availability. Under an assumption that all replicas of an object are managed on different peers in the DHT, peer and replica availability should be equal. Based on replicas managed locally, a peer can check if other replicas of an object are available as well. We are fully aware that absolute precision cannot be achieved. Thus, based on the confidence interval theory, we determine the number of probes that is sufficient for obtaining measurements

with an acceptable error. To additionally reduce the number of exchanged messages, probing is done in two steps. Every peer checks first the availability of a limited number of replicas. Along with the answer about replica availability, the peer gets all measurements known by the peer that has received the probing request. These measurements are made at different points in the past. Afterwards, the peer has enough information to calculate the peer availability precisely. A linear curve is fitted to the collected measurements using the Least Square method. Its value at the moment when the peer performs the measurement defines the current peer availability.

Finally, the replication protocol proposed in this thesis has been fully evaluated against the defined goals. The results have shown that it reaches, maintains, and recovers the pre-configured data availability and consistency in a DHT with an arbitrary and unknown peer availability. The guarantees are delivered both in cases of immutable and mutable data, even if the update rate applied is fairly artificial and high. Peer availability is measured with the expected precision, and the storage and communication overhead introduced match the models developed during the analysis phase.

6.2 Limitations

The self-adaptation of the proposed replication protocol relies upon the ability to measure actual peer availability with a high precision (i.e. the maximum allowed error δ with high probability). On the other hand, the defined measurement technique is based on the following assumptions:

- A peer manages not more than one replica of an object with a very high probability
- A peer manages enough replicas used as the basis for probing a sufficient number (depends on the maximum allowed error δ) of randomly chosen replicas

If they are not fulfilled, the obtained values, respectively delivered guarantees will not be achieved, i.e. the error will be higher. During the construction of the evaluation scenarios, we have seen that the first assumption does not hold if the DHT is smaller in size (less than 100 peers). As expected in the BRICKS project, the number of peers should be significantly larger. Similar to this, DHTs already deployed in practice are usually much larger. Hence, the first assumption will not be violated in regular cases.

The validity of the second assumption depends on the number of replicas S managed by a peer, previously determined replication factor R, and the number of probes n needed for the requested precision. If $(S-1)R < n$, the peer is not able to select enough replicas for probing. Thus, it will refuse to perform the measurement at all, because the requested precision cannot guaranteed. If measurements cannot be made, adjustment of the replication factor cannot be carried out, and the requested guarantees cannot be delivered fully.

As the evaluation carried out in this thesis has shown, probing 30 randomly chosen replicas delivers fairly precise measurements. Even in a highly-available DHT, the number of replicas R per an object is usually greater than 5. Thus, as soon as there are 6 replicas in its storage, the

peer would be able to measure precisely. Usually, DHTs manage a high volume of data. The scenario defined in the BRICKS project should keep a large set of metadata in the decentralized XML store as well. Hence, the assumption will remain, and precision will not be influenced.

6.3 Future Research

The research presented in this thesis has proposed the approach that solves the problem described in Section 1.2. However, we should not stop here. Future research should address the detected drawbacks and issues that were not fully or not at all in the focus of the thesis:

Solving the detected limitations As stated in the previous Section, peer availability measurements could sometimes be imprecise due to the lack of a sufficient number of replicas managed by a peer. In order to avoid high error in such a situation, the measurement mechanism should take into account the available knowledge about the underlying DHT topology, i.e. known peer IDs. They could be gathered from received or forwarded messages with the DHT. Later on, peer availability could be checked directly (e.g. pinging) along with the existing replica probing. The challenge is to develop a mechanism that allows a peer to gather as many peer IDs as possible in a fully decentralized way with moderate costs. At the same time, it should be investigated how this influences peers' privacy and what a strategy to protect it is.

Influence of history and cluster length on the measurement precision The thesis has described roughly how changing the history and cluster length influences the measured peer availability. To understand all effects fully, a detailed analysis is needed. The relationships between them and the time needed to reach/restore the requested guarantees should be found. In addition to that, it could be interesting to see if the cluster and history length could change during run-time, i.e. would this deliver better system performances.

Faster recovery from churns by predicting peer's behavior The measurement technique proposed in this thesis constructs a curve that fits probes done at different point in past. As the evaluation has demonstrated, it helps to follow churn trends and to get more accurate peer availability. The fitted curve allows us potentially to look into the future and see what the peer availability is going to be. It would be very interesting to see if this could be used for making recovery from churns faster. The main challenges are how to determine how far into the future we could look, and how much we could trust the predicted value.

Protocol evaluation during multiple, arbitrary churns The evaluation done in this thesis has shown that we are able to recover and/or keep the requested guarantees during or shortly after the end of the churn. Further evaluation should take into consideration scenarios where multiple, arbitrary churns appear in the DHT. It should be discovered which churn patterns are not handled efficiently. If possible, potential improvements should be suggested.

Delivering the requested guarantees during DHT bootstrap The work done in this thesis assumes that a DHT of a given size is deployed first and then an amount of data is inserted into it. It would be interesting to observe how the protocol behaves if applied to a DHT that is bootstrapped from scratch and whose size (i.e. number of participating peers) and amount of managed data increase over time. Experiments should be done with various distributions that describe how the number of participating peers or the amount of managed data change over time.

Improving performance of data access The research presented has not focused on optimizing the data access times. The data placement has been been left to the underlying DHT implementation. Future research should investigate if the replication protocol can optimize it for common data access patterns. The challenge is to define a strategy that is independent from the routing mechanism of the underlying DHT implementation.

Bibliography

[ABC+02] Atul Adya, William J. Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R. Douceur, Jon Howell, Jacob R. Lorch, Marvin Theimer, and Roger P. Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment, December 2002.

[Abe01] Karl Aberer. P-Grid: A self-organizing access structure for P2P information systems. *Lecture Notes in Computer Science*, 2172, 2001.

[ADN+96] Thomas E. Anderson, Michael D. Dahlin, Jeanna M. Neefe, David A. Patterson, Drew S. Roselli, and Randolph Y. Wang. Serverless network file systems. *ACM Trans. Comput. Syst.*, 14(1):41–79, 1996.

[AH00] Eytan Adar and Bernardo Huberman. Free riding on Gnutella. *First Monday*, 5(10), October 2000. `http://www.firstmonday.org/issues/issue5_10/adar/`.

[AL01] Paul Albitz and Cricket Liu. *DNS and BIND, Fourth Edition.* O'Reilly & Associates, April 2001.

[Bab06] Arne Babenhauserheide. Why gnutella scales quite well, November 2006. `http://basis.gnufu.net/gnufu/index.php/Why_Gnutella_scales_quite_well`.

[BBP+02] Dmitry Brodsky, Alex Brodsky, Jody Pomkoski, Shihao Gong, Michael J. Feeley, and Norman C. Hutchinson. Using file-grain connectivity to implement a peer-to-peer file system. In *Proceedings of 21st IEEE Symposium on Reliable Distributed Systems (SRDS'02)*, pages 318–323, October 2002.

[BBST01] Christopher Batten, Kenneth Barr, Arvind Saraf, and Stanley Treptin. pStore: A aecure peer-to-peer backup system. Technical Report MIT-LCS-TM-632, MIT Laboratory for Computer Science, December 2001.

[BG84] Philip A. Bernstein and Nathan Goodman. An algorithm for concurrency control and recovery in replicated distributed databases. *ACM Trans. Database Syst.*, 9(4):596–615, 1984.

[BHG87] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison Wesley, 1987.

[Bit06] BitTorrent Inc. *BitTorrent Homepage*, 2006. `http://www.bittorrent.com/`.

[BL95] Donald A. Berry and Bernard W. Lindgren. *Statistics: Theory and Methods*. Duxbury Press, October 1995.

[BMSV02] Ranjita Bhagwan, David Moore, Stefan Savage, and Geoffrey M. Voelker. Replication strategies for highly available peer-to-peer storage systems. In *Proceedings of FuDiCo: Future Directions in Distributed Computing*, June 2002.

[BRI04] BRICKS Consortium. *BRICKS - Building Resources for Integrated Cultural Knowledge Services (IST 507457)*, 2004. `http://www.brickscommunity.org/`.

[BSJBR80] Philip A. Bernstein, David W. Shipman, and Jr. James B. Rothnie. Concurrency control in a system for distributed databases (sdd-1). *ACM Trans. Database Syst.*, 5(1):18–51, 1980.

[BSV03] Ranjita Bhagwan, Stefan Savage, and Goeffrey M. Voelker. Understanding availability. In *2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, February 2003.

[BTC+04] Ranjita Bhagwan, Kiran Tati, Yu-Chung Cheng, Stefan Savage, and Geoffrey M. Voelker. Total recall: System support for automated availability management. In *First ACM/Usenix Symposium on Networked Systems Design and Implementation*, pages 337–350, March 29-31 2004.

[BWDD02] Peter Backx, Tim Wauters, Bart Dhoedt, and Piet Demeester. A comparison of peer-to-peer architectures. In *Eurescom Summit*, 2002.

[Cam97] Richard Campbell. *Managing AFS: The Andrew File System*. Prentice-Hall, 1997.

[CAMN03] Francisco Matias Cuenca-Acuna, Richard P. Martin, and Thu D. Nguyen. Autonomous replication for high availability in unstructured p2p systems. *srds*, 00:99, 2003.

[CD89] George F. Coulouris and Jean Dollimore. *Distributed Systems - Concepts and Design*. Addison Wesley, 1989.

[CHKS91] Stefano Ceri, Maurice A. W. Houtsma, Arthur M. Keller, and Pierangela Samarati. A classification of update methods for replicated databases. Technical report, Stanford, CA, USA, 1991.

[CMH+02] Ian Clarke, Scott G. Miller, Theodore W. Hong, Oskar Sandberg, and Brandon Wiley. Protecting free expression online with freenet. *IEEE Internet Computing*, 6(1):40–49, 2002.

[CMN02] Landon P. Cox, Christopher D. Murray, and Brian D. Noble. Pastiche: Making backup cheap and easy. In *OSDI '02: Proceedings of the 5th symposium on Operating systems design and implementation*, pages 285–298, New York, NY, USA, 2002. ACM Press.

[CMP82] Stefano Ceri, Giancarlo Martella, and Giuseppe Pelagatti. Optimal file allocation in a computer network: a solution method based on the knapsack problem. *Computer Networks*, 6(5):345–357, 1982.

[Com75] Paul G. Comba. Needed: Distributed control. In Douglas S. Kerr, editor, *Proceedings of the International Conference on Very Large Data Bases, September 22-24, 1975, Framingham, Massachusetts, USA*, pages 364–375. ACM, 1975.

[CP92] Shu-Wie Chen and Calton Pu. A structural classification of integrated replica control mechanisms. Technical report, New York, NY, USA, 1992.

[CSWH01] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. Freenet: A distributed anonymous information storage and retrieval system. *Lecture Notes in Computer Science*, 2009:46–66, 2001.

[Dav84] Susan B. Davidson. Optimism and consistency in partitioned distributed database systems. *ACM Trans. Database Syst.*, 9(3):456–481, 1984.

[DFM01] Roger Dingledine, Michael J. Freedman, and David Molnar. The free haven project: distributed anonymous storage service. In *International workshop on Designing privacy enhancing technologies*, pages 67–95, New York, NY, USA, 2001. Springer-Verlag New York, Inc.

[DGMS85] Susan B. Davidson, Hector Garcia-Molina, and Dale Skeen. Consistency in a partitioned network: a survey. *ACM Computing Surveys (CSUR)*, 17(3):341–370, 1985.

[DHA03] Anwitaman Datta, Manfred Hauswirth, and Karl Aberer. Updates in highly unreliable, replicated peer-to-peer systems. In *ICDCS '03: Proceedings of the 23rd International Conference on Distributed Computing Systems*, page 76, Washington, DC, USA, 2003. IEEE Computer Society.

[DKK+01] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, pages 202–215. ACM Press, 2001.

[DZDS03] Frank Dabek, Ben Zhao, Peter Druschel, and Ion Stoica. Towards a common api for structured peer-to-peer overlays. In *2nd International Workshop on Peer-to-Peer Systems*, February 2003.

[EJ01] Donald E. Eastlake and Paul E. Jones. US secure hash algorithm 1 (SHA1) (RFC 3174), September 2001. `ftp://ftp.ripe.net/rfc/rfc3174.txt`.

[Esw74] Kapali P. Eswaran. Placement of records in a file and file allocation in a computer. In *IFIP Congress*, pages 304–307, 1974.

[Far02] Farsite. Federated, available, and reliable storage for an incompletely trusted environment, 2002. `http://research.microsoft.com/Farsite`.

[Fre01] Freenet. *The FreeNet Project Homepage*, 2001. `http://www.freenetproject.org/`.

[Fre03] FreeHaven. *The Free Haven Project*, 2003. `http://freehaven.net/`.

[GHB81] Hector Garcia-Holina and Daniel Barbara. The cost of data replication. In *SIGCOMM '81: Proceedings of the seventh symposium on Data communications*, pages 193–198, New York, NY, USA, 1981. ACM Press.

[GKL+05a] Thomer M. Gil, Frans Kaashoek, Jinyang Li, Robert Morris, and Jeremy Stribling. *P2PSim: a Simulator for Peer-to-peer Protocols*, 2005. `http://pdos.csail.mit.edu/p2psim/`.

[GKL+05b] Thomer M. Gil, Frans Kaashoek, Jinyang Li, Robert Morris, and Jeremy Stribling. *P2PSim: a Simulator for Peer-to-peer Protocols*, 2005. `http://http://www.oversim.org/`.

[Gnu01] Gnutella. *Gnutella Homepage*, 2001. `http://www.gnutella.com/`.

[GSBK04] Vijay Gopalakrishnan, Bujor Silaghi, Bobby Bhattacharjee, and Pete Keleher. Adaptive replication in peer-to-peer systems. In *The 24th International Conference on Distributed Computing Systems (ICDCS'04)*, pages 360–369, March 2004.

[HA87] Mark Horton and Rick Adams. Standard for interchange of USENET messages, Internet RFC1036. 1987.

[HBH96] Abdelsalam A. Helal, Bharat K. Bhargava, and Abdelsalam A. Heddaya. *Replication Techniques in Distributed Systems*. Kluwer Academic Publishers, Norwell, MA, USA, 1996.

[JJMV06] Mark Jelasity, Gian Paolo Jesi, Alberto Montresor, and Spyros Voulgaris. *PeerSim P2P Simulator*, 2006. `http://peersim.sourceforge.net/`.

[JPPMAK03] Ricardo Jiménez-Peris, M. Patiño-Marténez, Gustavo Alonso, and Bettina Kemme. Are quorums an alternative for data replication? *ACM Trans. Database Syst.*, 28(3):257–294, 2003.

[KaZ02] KaZaA. *KaZaA Homepage*, 2002. `http://www.kazaa.com/`.

[KBC+00] John Kubiatowicz, David Bindel, Yan Chen, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westly Weimer, Christopher Wells, and Ben Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proceedings of ACM ASPLOS*. ACM, November 2000.

[Kne04] Predrag Knežević. Towards a reliable peer-to-peer xml database. In Wolfgang Lindner and Andrea Perego, editors, *Proceedings ICDE/EDBT Joint PhD Workshop 2004*, pages 41–50, P.O. Box 1527, 71110 Heraklion, Crete, Greece, March 2004. Crete University Press.

[KWR05] Predrag Knežević, Andreas Wombacher, and Thomas Risse. Highly available DHTs: Keeping data consistency after updates. In *Proceedings of International Workshop on Agents and Peer-to-Peer Computing (AP2PC)*, 2005.

[KWR06a] Predrag Knežević, Andreas Wombacher, and Thomas Risse. DHT-based self-adapting replication protocol for achieving high data availability. In *The International Conference on Signal-image Technology and Internet–based Systems (SITIS)*, December 2006.

[KWR06b] Predrag Knežević, Andreas Wombacher, and Thomas Risse. Managing and recovering high data availability in a dht under churn. In *The 2nd International Conference on Collaborative Computing: Networking, Applications and Worksharing*, November 2006.

[KWR07] Predrag Knežević, Andreas Wombacher, and Thomas Risse. High data availability and consistency for distributed hash tables depl in dynamic peer-to-peer communities. In *ICDE (submitted)*, 2007.

[KWRF05] Predrag Knežević, Andreas Wombacher, Thomas Risse, and Peter Fankhauser. Enabling high data availability in a DHT. In *Grid and Peer-to-Peer Computing Impacts on Large Scale Heterogeneous Distributed Database Systems (GLOBE'05)*, 2005.

[LCL04] W. K. Lin, D. M. Chiu, and Y. B. Lee. Erasure code replication revisited. In *Fourth International Conference on Peer-to-Peer Computing*, pages 90–97, August 2004.

[LJ04] Shiding Lin and Chao Jin. Repstore: A self-managing and self-tuning storage backend with smart bricks. In *ICAC '04: Proceedings of the First International Conference on Autonomic Computing (ICAC'04)*, pages 122–129, Washington, DC, USA, 2004. IEEE Computer Society.

[MKL+02] Dejan Milojičić, Vana Kalogeraki, Rajan Lukose, Kiran Nagaraja, Jim Pruyne, Bruno Richard, Sami Rollins, and Zhichen Xu. Peer-to-peer computing. Technical report, HP, 2002. `http://www.hpl.hp.com/techreports/2002/HPL-2002-57.pdf`.

[MM02] Petar Maymounkov and David Mazières. Kademlia: A peer-to-peer information system based on the xor metric. In *IPTPS '01: Revised Papers from the First International Workshop on Peer-to-Peer Systems*, pages 53–65, London, UK, 2002. Springer-Verlag.

[MMGC02] Athicha Muthitacharoen, Robert Morris, Thomer Gil, and Benjie Chen. Ivy: A read/write peer-to-peer file system. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, Massachusetts, December 2002.

[MNR02] Dahlia Malkhi, Moni Naor, and David Ratajczak. Viceroy: a scalable and dynamic emulation of the butterfly. In *PODC '02: Proceedings of the twenty-first annual symposium on Principles of distributed computing*, pages 183–192, New York, NY, USA, 2002. ACM Press.

[MS03] William Mendenhall and Terry L. Sincich. *A Second Course in Statistics: Regression Analysis*. Prentice Hall, 2003.

[Nap01] Napster. *Napster Homepage*, 2001. `http://www.napster.com/`.

[NB77] Erich J. Neuhold and Horst Biller. Porel: A distributed data base on an inhomogeneous computer network. In *VLDB*, pages 380–395, 1977.

[PGK88] David A. Patterson, Garth Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (raid). In *SIGMOD '88: Proceedings of the 1988 ACM SIGMOD international conference on Management of data*, pages 109–116, New York, NY, USA, 1988. ACM Press.

[Pla06] PlanetLab. An open platform for developing, deploying, and accessing planetary-scale services, 2006. `http://www.planet-lab.org`.

[PNP88] Calton Pu, Jerre D. Noe, and Andrew Proudfoot. Regeneration of replicated objects: A technique and its eden implementation. *IEEE Trans. Softw. Eng.*, 14(7):936–945, 1988.

[PRR97] C. Greg Plaxton, Rajmohan Rajaraman, and Andréa W. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *SPAA '97: Proceedings of the ninth annual ACM symposium on Parallel algorithms and architectures*, pages 311–320, New York, NY, USA, 1997. ACM Press.

[PST+97] Karin Petersen, Mike J. Spreitzer, Douglas B. Terry, Marvin M. Theimer, and Alan J. Demers. Flexible update propagation for weakly consistent replication. In *SOSP '97: Proceedings of the sixteenth ACM symposium on Operating systems principles*, pages 288–301, New York, NY, USA, 1997. ACM Press.

[RD01a] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *Lecture Notes in Computer Science*, 2218, 2001.

[RD01b] Antony Rowstron and Peter Druschel. Storage management and caching in past, a large-scale, persistent peer-to-peer storage utility. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 188–201, New York, NY, USA, 2001. ACM Press.

[RFH+01] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content-addressable network. In *Computer Communication Review*, volume 31, pages 161–172. Dept. of Elec. Eng. and Comp. Sci., University of California, Berkeley, 2001.

[Rit01] Jordan Ritter. Why gnutella cannot scale. no really, February 2001. `http://www.darkridge.com/~jpr5/doc/gnutella.html`.

[Riv92] Ronald L. Rivest. The MD5 message-digest algorithm (RFC 1321), 1992. `ftp://ftp.ripe.net/rfc/rfc1321.txt`.

[RKW04] Thomas Risse, Predrag Knežević, and Andreas Wombacher. P2P evolution: From file-sharing to decentralized workflows. *it-Information Technology*, 4:193–199, 2004.

[RL05] Rodrigo Rodrigues and Barbara Liskov. High availability in DHTs: Erasure coding vs. replication. In *IPTPS '05: International Workshop on Peer-to-Peer Systems*, February 24-25 2005.

[RS04] Venugopalan Ramasubramanian and Emin Gün Sirer. Beehive: O(1) lookup performance for power-law query distributions in peer-to-peer overlays. In *Networked System Design and Implementation (NSDI)*, March 2004.

[RWE+01] Sean Rhea, Chris Wells, Patrick Eaton, Dennis Geels, Ben Zhao, Hakim Weatherspoon, and John Kubiatowicz. Maintenance-free global data storage. *IEEE Internet Computing*, 5(5):40–49, 2001.

[SGG02a] Stefan Saroiu, P. Krishna Gummadi, and Steven D. Gribble. A measurement study of peer-to-peer file sharing systems. In *Proceedings of Multimedia Computing and Networking 2002 (MMCN '02)*, San Jose, CA, USA, January 2002.

[SGG02b] Stefan Saroiu, P. Krishna Gummadi, and Steven D. Gribble. A measurement study of peer-to-peer file sharing systems. In *Proceedings of Multimedia Computing and Networking 2002 (MMCN '02)*, San Jose, CA, USA, January 2002.

[SGK+85] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. The design and implementation of the sun network file system. In *In Proceedings Usenix Conference*, Portland, Ore, USA, 1985.

[SKKM02] Yasushi Saito, Christos Karamanolis, Magnus Karlsson, and Mallik Mahalingam. Taming aggressive replication in the pangaea wide-area file system. *SIGOPS Oper. Syst. Rev.*, 36(SI):15–30, 2002.

[SMK+01] Ion Stoica, Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. In *Proceedings of the 2001 ACM SIGCOMM Conference*, pages 149–160, 2001.

[SN77] Michael Stonebraker and Erich J. Neuhold. A distributed database version of INGRES. In *Berkeley Workshop on Distributed Data Management and Computer Networks*, pages 19–36, May 1977.

[Sto79] Michael Stonebraker. Concurrency control and consistency of multiple copies of data in distributed ingres. *IEEE Trans. Software Eng.*, 5(3):188–194, 1979.

[TD03] Nyik San Ting and Ralph Deters. 3LS — a peer-to-peer network simulator. In *P2P '03: Proceedings of the 3rd International Conference on Peer-to-Peer Computing*, page 212, Washington, DC, USA, 2003. IEEE Computer Society.

[Tho79] Robert H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Trans. Database Syst.*, 4(2):180–209, 1979.

[Tho05] Clive Thompson. The bittorrent effect. *Wired Magazine*, Issue 13.01, January 2005.

[TTP+95] Douglas B. Terry, Marvin M. Theimer, Karin Petersen, Alan J. Demers, Mike J. Spreitzer, and Carl H. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. In *SOSP '95: Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 172–182, New York, NY, USA, 1995. ACM Press.

[Uni06] Rice University. *FreePastry - Open-source Pastry Implementation*, 2006. http://freepastry.org/FreePastry/.

[W3C99] W3C. *XML Path Language (XPath) Version 1.0*, 1999. http://www.w3c.org/TR/xpath.

[W3C02] W3C. *Document Object Model*, 2002. http://www.w3.org/DOM/.

[W3C03] W3C. *XML Query*, 2003. http://www.w3c.org/XML/Query.

[Wel00] Chris Wells. The OceanStore archive: Goals, structures, and self-repair. Technical Report U.C. Berkeley Masters Report, UC Berkeley, May 2000.

[Wik03a] Wikipedia. *eDonkey*, 2003. http://en.wikipedia.org/wiki/EDonkey_network.

[Wik03b] Wikipedia. *Overnet*, 2003. http://en.wikipedia.org/wiki/Overnet.

[Wik06] Wikipedia. *FastTrack Protocol*, 2006. http://en.wikipedia.org/wiki/FastTrack.

[WJH97] Ouri Wolfson, Sushil Jajodia, and Yixiu Huang. An adaptive data replication algorithm. *ACM Trans. Database Syst.*, 22(2):255–314, 1997.

[WK02] Hakim Weatherspoon and John Kubiatowicz. Erasure coding vs. replication: A quantitative comparison. In *IPTPS '01: Revised Papers from the First International Workshop on Peer-to-Peer Systems*, pages 328–338. Springer-Verlag, 2002.

[YAG05] Weishuai Yang and Nael Abu-Ghazaleh. Gps: A general peer-to-peer simulator and its use for modeling bittorrent. In *MASCOTS '05: Proceedings of the 13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 425–434, Washington, DC, USA, 2005. IEEE Computer Society.

[YGM03] Beverly Yang and Hector Garcia-Molina. Designing a super-peer network. *ICDE*, 00:49, 2003.

[Yu00] Haifeng Yu. Tact: tunable availability and consistency tradeoffs for replicated internet services (poster session). *SIGOPS Oper. Syst. Rev.*, 34(2):40, 2000.

[YV02] Haifeng Yu and Amin Vahdat. Design and evaluation of a conit-based continuous consistency model for replicated services. *ACM Trans. Comput. Syst.*, 20(3):239–282, 2002.

[YV05] Haifeng Yu and Amin Vahdat. Consistent and automatic replica regeneration. *ACM Transactions on Storage*, 1(1):3–37, 2005.

[ZHR+04] Ben Y. Zhao, Ling Huang, Sean C. Rhea, Jeremy Stribling, Anthony D. Joseph, and John D. Kubiatowicz. Tapestry: A global-scale overlay for rapid service deployment. *IEEE Journal on Selected Areas in Communications (J-SAC)*, 22(1):41–53, January 2004.

[ZKJ01] Ben Y. Zhao, John D. Kubiatowicz, and Anthony D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, UC Berkeley, April 2001.

Curriculum Vitae

Personal Data

Full Name	Predrag Knežević
Date of Birth	02.01.1974
Place of Birth	Zrenjanin, Serbia
Nationality	Serbian

Education

2000 - 2001	PhD preparation school, EPFL, Lausanne, Switzerland
1993 - 1999	Graduate Engineer University in Belgrade, Serbia, School of Electrical Engineering, Computer science and informatics department. Average score 9,02/10
	Diploma topic: “Issues in FPGA-based Configurable Computer Design for Multimedia Applications and Operating Systems" (Note 10)
1989-1993	Gymnasium, Zrenjanin, Serbia

Awards

2000	Scholarship for EPFL PhD preparation school
1999	Scholarship from Serbian Ministry of Science for young researcher
1999	The best student of Computer science and informatics department, School of Electrical Engineering, winter/summer semester 1998/99

Professional Experience

12/2001 - present	Research Associate at Fraunhofer IPSI Institute, Darmstadt, Germany
12/1997 - 10/2000	Chief engineer at Media house B92, Belgrade, Serbia
03/2000 - 10/2000	System administrator at AAEN, Belgrade, Serbia
07/1999 - 05/2000	Software developer at Belgrade University Computing Center, Serbia
04/1998 - 01/1999	Hardware designer at School of Electrical Engineering, Belgrade, Serbia
12/1998 - 06/1999	Hardware designer at Virtual Computer Company, Belgrade, Serbia
01/1997 - 12/1997	Journalist at Computer magazine “Micro Racunari”, Belgrade, Serbia
01/1995 - 12/1997	Journalist at Computer magazine “Racunari”, Belgrade, Serbia