UniWiki: A Collaborative P2P System for Distributed Wiki Applications

Gérald Oster and Pascal Molli
Nancy-Université, INRIA
France
{oster,molli}@loria.fr

Sergiu Dumitriu
XWiki SAS, Nancy-Université
France
sergiu@xwiki.com

Ruben Mondejar
Universitat Rovira i Virgili
Spain
ruben mondejar@urv.cat

Abstract

The ever growing request for digital information raises the need for content distribution architectures providing high storage capacity, data availability and good performance. While many simple solutions for scalable distribution of quasi-static content exist, there are still no approaches that can ensure both scalability and consistency for the case of highly dynamic content, such as the data managed inside wikis. In this paper, we propose a peer to peer solution for distributing and managing dynamic content, that combines two widely studied technologies: distributed hash tables (DHT) and optimistic replication. In our “universal wiki” engine architecture (UniWiki), on top of a reliable, inexpensive and consistent DHT-based storage, any number of front-ends can be added, ensuring both read and write scalability. The implementation is based on a Distributed Interception Middleware, thus separating distribution, replication, and consistency responsibilities, and also making our system usable by third party wiki engines in a transparent way. Uniwiki is suitable for large-scale scenarios.

1. Introduction

Currently, Peer to peer (P2P) networks mainly distribute immutable contents. We aim at making use of their characteristics for distributing dynamic, editable content. More precisely, we propose to distribute updates on this content and manage collaborative editing on top of such a peer to peer network. We are convinced that, if we can deploy a group editor framework on a P2P network, we open the way for P2P content editing: a wide range of existing collaborative editing applications, such as CVS and Wikis, can be redeployed on P2P networks, and thus benefit from the availability improvements, the performance enhancements and the censorship resilience of P2P networks.

Our architecture targets heavy-load systems, that must serve a huge number of requests. An illustrative example is Wikipedia, the collaborative encyclopedia that has collected, until now, over 11 million articles in more than 260 languages. It currently registers at least 350 million page requests per day, and over 300,000 changes are made daily. To handle this load, Wikipedia needs a costly infrastructure, for which hundreds of thousands of dollars are spent every year. A P2P massive collaborative editing system would allow to distribute the service and share the cost of the underlying infrastructure.

Few approaches have been proposed to deploy a collaborative editing systems over a peer-to-peer network. Unfortunately, they either require a total replication of content, requiring all wiki pages are replicated at each peer, or they provide only a basic reconciliation mechanism for concurrent updates that is not suitable for collaborative authoring.

This paper presents the design and the first experimentations of a wiki architecture that:

- is able to store huge amounts of data,
- runs on commodity hardware by making use of peer to peer networks,
- does not have any single point of failure, or even a relatively small set of points of failure,
- is able to handle concurrent updates, ensuring eventual consistency.

To achieve these objectives, our system relies on the results of two intensively studied research domains, distributed hash tables (DHT) and optimistic replication [1]. At the storage system level, we use DHTs, which have been proved [2] as quasi-reliable even in test cases with a high degree of churning and network failures. However, DHTs alone are not designed for supporting consistently unstable content, with a high rate of modifications, as it is the case with the content of a wiki. Therefore, instead of the actual data, our
system stores in each DHT node operations, more precisely the list of changes that produce the current version of a wiki document. It is safe to consider these changes as the usual static data stored in DHTs, given that an operation is stored in a node independently of other operations, and no actual modifications are performed on it. Because the updates can originate in various sources, concurrent changes of the same data might occur, and therefore different operation lists could be temporarily available at different nodes responsible for the same data. These changes need to be combined such that a plausible most recent version of the content is obtained. For this purpose, our system uses an optimistic consistency maintenance algorithm, WOOT [3], which guarantees eventual consistency, causal consistency and preserves effects of concurrent modifications.

To reduce the effort needed for the implementation, and to make our work available to existing wiki applications, we built our system using a distributed interception middleware called Damon [4]. Thus, we were able to reuse existing implementations for all the components needed, and integrate our method transparently.

First, the architecture of the UniWiki system and its related algorithms are described in Section 2. Then, an implementation of this system is presented in Section 3. Related approaches are quickly discussed in Section 4. Finally, our conclusions and future work are presented in Section 5.

2. The UniWiki Architecture

2.1. Overview

The central idea of our paper is to use a DHT system as the storage mechanism, with updates handled by running the reconciliation algorithm (WOOT) directly in the DHT nodes. As we will see, this integration can be easily performed by changing the behavior of the basic put and get methods provided by a DHT.

As depicted in Figure 1 our system consists of a DHT network, responsible for data storage, and wiki front-ends, responsible for handling client requests.

The DHT storage network is a hybrid network of dedicated servers and commodity systems donated by users, responsible for hosting all the data inside the wiki in a peer to peer manner. As in any DHT, each peer is assigned responsibility for a chunk of the key space to which resources are mapped. In our context, each peer is therefore responsible for a part of the wiki content. Each wiki resource is stored on a peer in the form of a WOOT model. This allows to tolerate concurrent updates, occurring either when parallel editions arise from different access points, or when temporary partitions of the network are merged back. The WOOT algorithm ensures eventual consistency – convergence – on the replicated content stored by peers responsible for the same keys.

The wiki front-ends are responsible for handling client requests, by retrieving the WOOT models for the requested page from the DHT, reconstructing the wiki content, rendering it into HTML, and finally returning the result to the client. In case of an update request, the front-end computes the textual differences corresponding to the changes performed by the user, transforms them into WOOT operations, then sends them to the DHT to be integrated, stored and propagated to the replicated peers. A wiki front-ends can be integrated as peer in the DHT network or runs on any server which offer a network connection to the DHT.

2.2. Data model and WOOT algorithm

In what follows we briefly describe the WOOT algorithm [3] which is responsible for maintaining consistency of replicated data regarding concurrent modifications.

WOOT deals with replicated data that are modeled as a sequence of blocks, or elements, where a block is a unit of the text, with a given granularity: either a character, word, sentence, line or paragraph. Once created, a block is enhanced with information concerning unique identification and precise placement even in a highly dynamic collaborative environment, thus becoming a W-character. When a block is deleted, the corresponding W-character is not deleted, but its visibility flag is set to false. This allows future blocks to be correctly inserted in the right place on a remote site, even if the preceding or following blocks have been deleted by a concurrent edit. W-characters corresponding to the same document form a partially ordered graph, a W-string, which is the model on which WOOT operates.
Every user action is transformed into a series of WOOT operations, which include references to the affected context (either predecessor and successor elements, either the element to be removed), then exchanged between peers. An operation always affects exactly one element: a block can be added to, or removed from the content. Upon receiving a remote operation, it is added to a queue and will be applied when it is causally ready, meaning that the preceding and following W-characters are part of the W-string in case of an insert operation, or the target W-character is part of the W-string in case of a delete operation.

When a document must be displayed, the WOOT algorithm computes a linearization of the W-string, and only the visible block are returned. This linearization is computed in such a way that the order of reception of the operations does not influence the result, as long as all the operations have been received.

In the context of UniWiki, enriched versions of wiki textual contents (W-String) are stored inside the DHT. Each peer is therefore responsible for a set of those WOOT page models. To update this model, WOOT operations computed from the user’s changes are inserted in the DHT, first by calls to the put method, and then by inherent DHT synchronization algorithms.

Determining the actual changes performed by the user requires not just the new version of the content, but also the original version on which he started editing. And, since transforming textual changes into WOOT operations requires knowing the WOOT identifiers corresponding to each block of text, this means that the front-end must remember, for each user and for each edited document, the model that generated the content sent to the user. However, not all the information in the complete WOOT page model is needed, but just the visible part of the linearized W-string, and only the id and actual block content. This simplified W-string – or, from a different point of view, enriched content – is called the WOOT page, and is the information that is returned by the DHT and stored in the front-end.

When sending the page to the client, the front-end further strips all the meta-information, since the client needs only the visible text. This is the plain wiki content which can be transformed into HTML markup, or sent as editable content back to the client.

2.3. Algorithms

2.3.1. Behavior of a Front-end Server. The behavior of a front server can be summarized as follows. When a client wants to look at a specific wiki page, the method onDisplay() is triggered on the front-end server. First, the server retrieves the corresponding WOOT page from the DHT, using the hashed URI of the requested wiki content as the key. The received WOOT page is transformed into plain wiki content, which is rendered as HTML if necessary, then sent to the client.

```java
onDisplay(contentURI) {  
  wootPage = dht.get(getHash(contentURI))  
  htmlContent = renderHTML(wikiContent)  
  return htmlContent
}
```

When a client requests a particular page in order to edit it, the method onEdit() is called on the front-end server. The WOOT page retrieved from the DHT is stored in the current editing session, so that the changes performed by the user can be properly detected (effects of user’s changes are reflected on the initial content displayed to him, and not on the most recent version, which might have changed). Then the wiki content is sent to the client for editing.

```java
onEdit(contentURI) {  
  wootPage = dht.get(getHash(contentURI))  
  wikiContent = extractContent(wootPage)  
  return wikiContent
}
```

When a client terminates his editing session and decides to save his modifications, the method onSave() is executed on the front-end server. First, the old version of the wiki content is extracted from the current editing session. A classical textual differences algorithm is used to compute modifications between this old version and the new version submitted by the client. These modifications are then mapped on the old version of the WOOT page in order to generate WOOT operations. Finally, these WOOT operations are submitted to the DHT.

```java
onSave(contentURI, newWikiContent) {  
  oldWootPage = servlet.session().get(contentURI)  
  oldWikiContent = extractContent(oldWootPage)  
  changes[] = diff(oldWikiContent, newWikiContent)  
  wootOps[] = ops2WootOps(changes[], oldWootPage)  
  dht.put(getHash(contentURI), wootOps[])  
}
```

2.3.2. Behavior of a DHT Peer. In order to comply to our architecture, the basic methods generally provided by a DHT peer have to be updated. Their behaviors differ from the basic behaviors provided by any DHT since the type of the value returned by a get request – a WOOT page – is not the same as the type of the value – a list of WOOT operations – stored by a put request.

When a get request is received by a DHT peer (which means that it is responsible for the wiki content
identified by the targeted key), the method `onGet()` is executed. The WOOT page model corresponding to the key is retrieved from the local storage, the simplified WOOT page is extracted, and then sent back to the requester — generally, a front-end server.

```java
onGet(key)
    wootPageModel = wootStore.get(key)
    wootPage = extractVisiblePage (wootPageModel)
    return wootPage
```

When a DHT peer has to answer to a `put` request, the method `onPut()` is triggered. First, the targeted WOOT page model is retrieved from the local storage. Then, each operation received within the request is integrated in the actual WOOT page and logged in the history of that page.

```java
onPut(key, wootOps[])
    wootPageModel = wootStore.get(key)
    for (op in wootOps[])
        integrate (op, wootPageModel)
    wootPageModel.log(op)
```

Generally, DHTs provide a mechanism for recovering from abnormal situations such as transient or permanent failure of a peer, message drop on network, or simply new nodes joining. In such situations, after the execution of the standard mechanism that re-attribute the keys responsibility to peers, the method `onRecover()` is called for each key the peer is responsible for.

The goal of the `onRecover()` method is to reconcile the history of a specific wiki content with the current histories of that content stored at other peers responsible for the same key. By the usage of the WOOT integration algorithm, reconciliating histories will also ensure convergence on the replicated WOOT models.

The method starts by retrieving the targeted WOOT page model and its history. Then, a digest of all the operations contained in this history is computed. Further, the method `antiEntropy()` is called on another replica – another peer responsible for the same key – in order to retrieve operations that the peer could have missed. Finally, each missing operation is integrated in the WOOT model and is added to its history.

```java
onRecover(key)
    wootPageModel = wootStore.get(key)
    wootOps[] = wootPageModel.getLog()
    digests [] = digest (wootOps[])
    missingOps[] = getReplica (key ), antiEntropy ( digests [])
    for (op in missingOps[])
        integrate (op, wootPageModel)
    wootPageModel.log(op)
```

3. Implementation

Over the last few years, many DHT implementations have been released. These systems take advantage of the computing at the edge paradigm, where resources from any computer in the network can be used, and are normally made available. However, contrasting with the client-server model, where few powerful servers provide all the computing, and storage power to a much larger network of clients.

Nevertheless, such decentralized architectures introduce new issues which have to be taken care of. Some of these issues include how to deal with constant node joins and leaves, network heterogeneity, and many others. Also, another important issue is the development complexity of new applications on top of this kind of networks. For these reasons, we need a middleware platform that provides the necessary abstractions and mechanisms to construct distributed applications.

Following the ideas developed in previous sections, we want to implement our algorithms over a DHT system. In this line, we summarize the objectives of our implementation in three points:

- Modify the `put` and `get` DHT methods behavior.
- Establish replication strategies and handle replicas.
- Add consistency logic in these DHT mechanisms.

In this setting, another important aspect is that we want to change the behavior of the DHT system. To address this new problem, there are different options, but it is clearly a good scenario to apply distributed interception mechanisms. The benefits of this approach will be:

- Full control of the DHT mechanisms, including runtime adaptations.
- Decoupled architecture between wiki front-end and DHT sides.
- Legacy and transparency for wiki front-end applications.

Lastly, we propose the use of a distributed P2P interception middleware, called Damon [4] that fulfills the DHT, middleware, and interception requirements. Using this middleware, developers can implement and compose distributed interceptors in large-scale environments. When the local interception (source hook) is performed, these distributed interceptors trigger the remote calls via P2P routing abstractions. For more detailed information on Damon, please see [4]
3.1. UniWiki Implementation

Like traditional wiki application (i.e. Wikipedia), we have the local execution of our wiki front-end. This scenario is ideal to apply distributed interception, because we can intercept the local behavior to extend/compose the necessary concerns. Thereby, using Damon we are able to inject our algorithms of distribution, replication, and consistency transparently.

We now describe the UniWiki execution step by step as shown in Figure 2, focusing on the integration of the algorithms and the interaction of the different concerns. In this line, we analyze the context, and extract three main concerns that we need to implement: distribution, replication and consistency.

![Figure 2. UniWiki composition diagram.](image)

Obviously, distribution is the main concern, and in our solution uses the key-based routing abstraction. The replication concern is also based on P2P mechanisms, and follows the neighbors strategy. Finally, as we see in previous section, the consistency concern remains on the WOOT algorithm. In this implementation, it allows edition, patching, and merging of wiki pages, and it performs these operations via distribution calls interception.

3.1.1. Distribution. The starting point of this application is the wiki interface used by the existing wiki application. We therefore introduce the Wiki Interface source hook that intercepts the save, and load methods. Afterward, the Locator distributed interceptor is deployed and activated on all nodes of the UniWiki network. Its main objective is to locate the responsible node of the local insertions and requests.

These save executions are propagated using the put connector. Consequently, the remote calls are routed to the key owner node, by using their url to generate the key through the key-base routing.

Once the key has reached its destination, the registered connectors are triggered on the Storage instance running on the owner host. This distributed interceptor has already been activated on start-up on all nodes. For request case (get), the idea is basically the same, with the Storage receiving the remote calls.

Furthermore, it propagates later an asynchronous response using the return call via direct node routing. Finally, the get values are returned to the Locator originator instance, using their own connector.

Once we have the wiki page distribution running, we may add new functionalities as well. In this sense, we introduce new distributed interceptors in order to extend or modify the current application behavior in runtime. More specifically, these instances add new important services (replication and consistency) as shown right away.

3.1.2. Replication. When dealing with the save method case, we need to avoid any data storage problems which may be present in such dynamic environments as large-scale networks. Thus, data is not only to be stored on the owner node, because if this host leaves the network for any reason, its data would surely become unavailable. In order to address this problem, we activate the Replicator interceptor in runtime, which following a specific policy. The Replicator has a connector called onPut, which intercepts the Storage put requests from the Locator service in a transparent way.

Thus, when a wiki page insertion arrives to the Storage instance, this information is re-sent (replicate) to the ReplicaStore instances activated in the closest neighbors.

Last but not least, ReplicaStore distributed interceptors are continuously observing the state of the copies that they are keeping. If one of them detects that the original copy is not reachable, it re-inserts the object again, using a remote connector put, in order to replace the Locator remote call.

3.1.3. Consistency. Based on the WOOT framework, we create the Editor (situated on the front-end side) and the Integrator (situated on the back-end side) distributed interceptor, those intercept the DHT-based calls to perform the consistency behavior. Their task is the modification of the distribution behavior, adding the patch transformation in the edition phase, and the patch merging in the storage phase.

The Editor distributed interceptor owns a connector (edit) that intercepts the return remote calls from Storage to Locator instances. This mechanism stores the original value in a session. Obviously, in a parallel way, the Integrator prepares the received information to be rendered as a wiki page.
Later, if the page is modified, a save method triggers the put mechanism, where another connector (patch) transforms the wiki page into the patch information by using the saved session value.

In the back-side, the Integrator instance intercepts the put request, and merges the new patch information with the back-end contained information. The process is similar to the original behavior, but changing the wiki page with a consistent patch information.

In this setting, having multiple replicated copies, leads to inconsistencies. We use the antiEntropy technique, in order to recover a log of differences among each page and their respective replicas. Finally, the Integrator sends the necessary patches to be sure that all copies remain consistent.

4. Related Work

Few decentralized wiki engines have been proposed such as DistriWiki [5], Wooki [6], DTWiki [7], Piki [8], RepliWiki [9] or the one described in [10]. These approaches have one or more of the following drawbacks. They either require that wiki content is fully replicated on every peer which is not acceptable in the context of a huge wiki such as Wikipedia. Or, they require that contributors install a specific rich client application in order to physically join and participate in the peer-to-peer network, and those users have to use this application instead of a standard web browser to contribute or consult any wiki content. Or, they propose an unsatisfactory solution to concurrent modifications problems by either creating two distinct versions of the wiki page and delegating the merging task to users, or, by choosing a transactional approach and rejecting unelected concurrent contribution.

5. Conclusions and Future Work

In this paper we propose an efficient P2P system for storing distributed wikis, transparently extended to large-scale scenarios. In this line we present the algorithms, a prototype, and experiments. We combine two intensively studied technologies, each one addressing a specific aspect: distributed hash tables are used as the underlying storage and distribution mechanism, and WOOT ensures that concurrent changes are correctly propagated and merged for every replica.

We propose a completely decoupled architecture, where our solution is totally abstracted from the real wiki application. In our approach we define the storage behavior from the scratch, apply this behavior on an existing DHT library, and the wiki presentation and business logic is full provided by the wiki application. Therefore, for our implementation we use a distributed interception middleware over DHT-based networks, called Damon. Our prototype is freely available at http://uniwiki.sourceforge.net/.

In the near future, we plan to apply our approach to a more sophisticated front-end, the XWiki application server. Other future directions include studying how to manage security access on wiki applications deployed on our proposed peer-to-peer network, and how to perform search.

Acknowledgments

This work has been partially funded by the Spanish Ministry of Science and Innovation through project P2PGRID TIN2007-68050-C03-03.

References