UniWiki: A Collaborative P2P System for Distributed Wiki Applications

Gérald Oster, Pascal Molli, Sergiu Dumitriu, Rubén Mondéjar

UniWiki: A Collaborative P2P System for Distributed Wiki Applications

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

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

Rubén Mondéjar
Universitat Rovira i Virgili
Tarragona, 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. 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, as well as suitability for large-scale scenarios. The implementation is based on a Distributed Interception Middleware, thus separating distribution, replication, and consistency responsibilities, and also making our system transparently usable by third party wiki engines.

1. Introduction

Peer to peer (P2P) systems, relying on content replication at more than one node to ensure scalable distribution, can be seen as a very large distributed storage system, used mainly for hosting quasi-immutable content. 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 P2P 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 Wikis and software configuration management tools, 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 encyclopædia 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 system over a P2P network. Unfortunately, they either require a total content replication, with copies of every wiki page available 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 P2P 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 *operations* in each DHT node, 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 concurrents 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 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 inside the DHT nodes. Section 3 will explain how this integration can be easily performed by changing the behavior of the basic **put** and **get** methods provided by any 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.

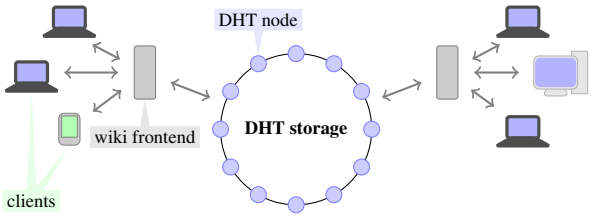

Figure 1. UniWiki architecture overview.

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 P2P 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**, explained in section 2.2. 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.

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 is 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 at 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, named a W-string, which is the model on which WOOT operates.

Every user change is transformed into a series of WOOT operations, which includes references to the affected context (either predecessor and successor elements, or 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 such 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 **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 made by the user requires not just the new version of the content, but also the original version on which the editing was performed. 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 view 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.

```
onDisplay(contentURI)
  wootPage = dht.get(getHash(contentURI))
  wikiContent = extractContent(wootPage)
  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.

```
onEdit(contentURI)
  wootPage = dht.get(getHash(contentURI))
  session.store(contentURI, wootPage)
  wikiContent = extractContent(wootPage)
  return wikiContent
```

When a client terminates the editing session and decides to save the new version, 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.

```
onSave(contentURI, newWikiContent)
  oldWootPage = 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 DTH 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.

```
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.

```
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-attributes 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 reconciliate 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 ensures convergence on the replicated WOOT models.

The method starts by retrieving the targeted WOOT page model and its history. Then, a digest of each 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, every missing operation is integrated in the WOOT model and is added to its history.

```
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)

onAntiEntropy(key, digests [])
  wootPageModel = wootStore.get(key)
  wootOps[] = wootPageModel.getLog()
  mydigests [] = digest (wootOps[])
  return notin (wootOps[], mydigests [], digests [])
```

3. Implementation

3.1. Context and Motivation

Nowadays, wikis are a popular concept, and many mature, fully featured wiki engines are publicly available. The storage has become a granted base, on which more advanced features are built, such as complex rights management, semi-structured and semantic data, advanced plugins, or support for scripting. Therefore, a completely new wiki engine, whose sole advantage is the scalability, would fail to properly fulfill many of the current wiki application requirements. Instead, we create a system that can be integrated transparently in existing wiki engines. Our implementation is driven by this transparency goal, and for achieving it, we rely on interception techniques (i.e. via Aspect Oriented Programming – AOP), by means of which existing behavior can be adapted.

A first such behavior change involves catching the calls issued by the wiki engine to its storage system and replacing them with calls to our distributed storage, as explained in 2.3.1. More advanced changes are needed on the DHT, for overriding the **put** and **get** behaviors, establishing replication strategies, and adding consistency logic among replicas.

Nevertheless, decentralized architectures introduce new issues which have to be taken care of, including how to deal with constant node joins and leaves, network heterogeneity, and, most importantly, the development complexity of new applications on top of this kind of network. For these reasons, we need a middleware platform that provides the necessary abstractions and mechanisms to construct distributed applications. 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.
- Transparency for legacy wiki front-end applications.

For satisfying the transparency and distributed interception requirements, we chose as the basis of our implementation the distributed P2P interception middleware Damon [4]. Using this middleware, developers can implement and compose distributed interceptors in large-scale environments. Such distributed interceptors, activated by local pointcuts (source hooks), trigger remote calls via P2P routing abstractions. A more detailed description of Damon middleware services is presented in [4].

3.2. UniWiki Implementation Overview

Like traditional wiki application (i.e. Mediawiki), 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.

Figure 2 presents the UniWiki source hook, where we aim at locally intercepting the typical wiki methods of store and retrieve (in this case we use a generic example), in order to distribute them remotely. In addition, the source hook solution helps to separate local interception, interceptor code, and the wiki interface. On the other hand, source hooks have other benefits such as a major level of abstraction, or degree of accessibility for distributed interceptors.

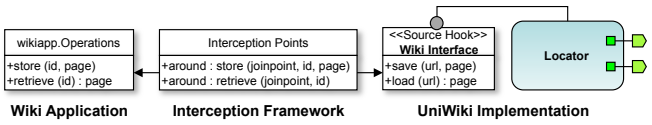

Figure 2. Wiki Source Hook Interface.

In this approach, integration with other wiki applications is quite simple and can be easily and transparently used for third party wiki applications.

We will further describe the UniWiki execution step by step, as shown in Figure 3, focusing on the integration of the algorithms and the interaction of the different concerns. We analyze the context, and extract three main concerns that we need to implement: distribution, replication, and consistency.

Obviously, distribution is the main concern, and our solution uses the key-based routing abstraction. The replication concern is also based on P2P mechanisms, and follows the neighbors strategy. Finally, as presented in a previous

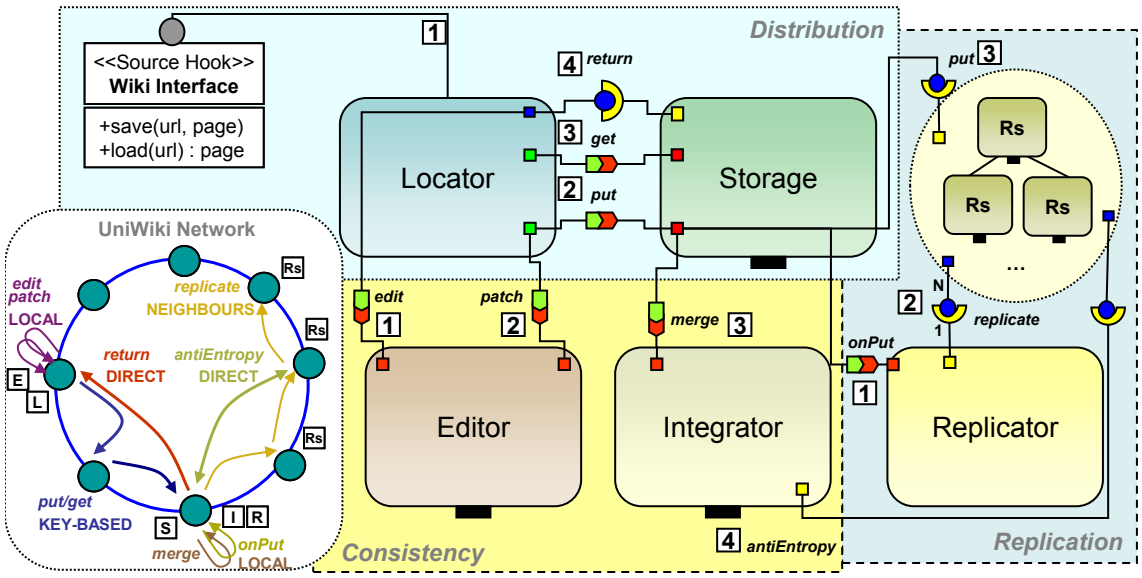

Figure 3. UniWiki composition diagram.

section, the consistency concern relies 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.

3.3. Distribution

The starting point of this application is the wiki interface. We introduce the **Wiki Interface** source hook that intercepts the **save** and **load** methods. The **Locator** distributed interceptor, which is deployed and activated at all nodes of the UniWiki network, locates the node responsible with 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 at all nodes. For request case (**get**), the idea is basically the same, with the Storage receiving the remote calls.

Furthermore, it later propagates 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 connectors.

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).

3.4. 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 at 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 follows a specific policy. The Replicator has a connector called **onPut**, which intercepts the Storage put requests from the Locator service transparently.

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, using a remote connector **put**, in order to replace the Locator remote call.

3.5. 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 interceptors, that intercept the DHT-based calls to ensure 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 data 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 **merge**s 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 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 transactional system described in [10]. These approaches have one or more of the following drawbacks: they require that the wiki content is fully replicated on every peer, which is not acceptable in the context of a huge wiki; they require that contributors install a specific rich client application in order to physically join and participate in the P2P network, and those users have to use this application instead of a standard web browser to contribute or consult any wiki content; 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 contributions.

5. Conclusions and Future Work

We presented the prototype of an efficient P2P system for transparently distributing the storage of wiki applications, which allows their extension to large-scale scenarios.

The transparency is ensured by a completely decoupled architecture, where our solution is totally abstracted from the real wiki application. At the storage level, 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 define the storage behavior from the scratch and apply it on an existing DHT library. A distributed interception middleware over DHT-based networks, called Damon, ensures the communication with the wiki application, which provides the presentation and business logic.

Validations of the UniWiki prototype are in progress on both Grid'5000 and PlanetLab testbeds. We are conducting experiments with real data from Wikipedia, consisting of almost 6 million entries.

The initial prototype, freely available at http://uniwiki.sourceforge.net/, was developed as a proof of concept project, using a simple wiki engine. We are currently working on refining the implementation, so that it can be fully applied on wikis with more complex storage requirements, such as XWiki or JSPWiki. Other future directions include abstracting the consistency maintenance concern, in order to be able to include other algorithms than WOOT, and approaching the management of security access and search on wiki applications deployed on our P2P network.

Acknowledgment

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

References

[1] Y. Saito and M. Shapiro, "Optimistic Replication," *ACM Computing Surveys*, vol. 37, no. 1, pp. 42–81, 2005.

[2] S. Ktari, M. Zoubert, A. Hecker, and H. Labiod, "Performance Evaluation of Replication Strategies in DHTs under Churn," in *Proceedings of the 6th International Conference on Mobile and Ubiquitous Multimedia - MUM 2007*. Oulu, Finland: ACM Press, Dec. 2007, pp. 90–97.

[3] G. Oster, P. Urso, P. Molli, and A. Imine, "Data Consistency for P2P Collaborative Editing," in *Proceedings of the ACM Conference on Computer-Supported Cooperative Work - CSCW 2006*. Banff, Alberta, Canada: ACM Press, Nov. 2006, pp. 259–267.

[4] R. Mondéjar, P. García, C. Pairot, P. Urso, and P. Molli, "Designing a Runtime Distributed AOP Composition Model," in *24th Annual ACM Symposium on Applied Computing - SAC 2009*. Honolulu, Hawaii, USA: ACM Press, Mar. 2009, pp. 539–540.

[5] J. C. Morris, "DistriWiki: A Distributed Peer-to-Peer Wiki Network," in *Proceedings of the 2007 International Symposium on Wikis - WikiSym 2007*. New York, NY, USA: ACM Press, 2007, pp. 69–74.

[6] S. Weiss, P. Urso, and P. Molli, "Wooki: a P2P Wiki-based Collaborative Writing Tool," *Lecture Notes In Computer Science*, vol. 4831, no. 1005, pp. 503–512, Dec. 2007.

[7] B. Du and E. A. Brewer, "DTwiki: A Disconnection and Intermittency Tolerant Wiki," in *Proceeding of the 17th International Conference on World Wide Web - WWW 2008*. New York, NY, USA: ACM Press, 2008, pp. 945–952.

[8] P. Mukherjee, C. Leng, and A. Schürr, "Piki - A Peer-to-Peer based Wiki Engine," *Proceedings of the IEEE International Conference on Peer-to-Peer Computing - P2P 2008*, vol. 0, pp. 185–186, 2008.

[9] "RepliWiki – A Next Generation Architecture for Wikipedia," http://isr.uncc.edu/repliwiki/.

[10] S. Plantikow, A. Reinefeld, and F. Schintke, "Transactions for Distributed Wikis on Structured Overlays," *Lecture Notes in Computer Science*, vol. 4785, pp. 256–267, Oct. 2007.