Building a collaborative peer-to-peer wiki system on a structured overlay

Gérald Oster, Rubén Mondéjar, Pascal Molli, Sergiu Dumitriu

Building a Collaborative Peer-to-Peer Wiki System on a Structured Overlay

Gérald Oster[a], Rubén Mondéjar[b], Pascal Molli[a], Sergiu Dumitriu[c]

[a]*Score Team, Université de Lorraine, Université Henri Poincaré, INRIA, LORIA, France*
[b]*Department of Computer Science and Mathematics, Universitat Rovira i Virgili, Spain*
[c]*XWiki SAS & Score Team, Université de Lorraine, Université Henri Poincaré, INRIA, LORIA, France*

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 HashTables (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 Damon, a distributed AOP middleware, thus separating distribution, replication, and consistency responsibilities, and also making our system transparently usable by third party wiki engines. Finally, UniWiki has been proved viable and fairly efficient in large-scale scenarios.

Key words:
Collaborative editing, Peer-to-peer, Distributed Interception, Optimistic replication

1. Introduction

Peer-to-peer (P2P) systems, which account for a significant part of all internet traffic, rely on content replication at more than one node to ensure scalable distribution. This approach can be seen as a very large distributed storage system, which has many advantages, such as resilience to censorship, high availability, virtually unlimited storage space [2].

Currently, 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 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 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 [40], the collaborative encyclopædia that has collected, until now, over 13,200,000 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 [42]. To handle this load, Wikipedia needs a costly infrastructure [3], for which hundreds of thousands of dollars are spent every year [11]. A P2P massive collaborative editing system would allow to distribute the service and share the cost of the underlying infrastructure.

Existing approaches to deploy a collaborative system on a distributed network include Wooki [38], DistriWiki [20], RepliWiki [29], Scalaris [32], Distributed Version Control systems [1, 13, 43], DTWiki [8] and Piki [21]. Several drawback prevent these systems from being used in our target scenario. They either require total replication of content, requiring all wiki pages are replicated at all nodes, or do not provide support for all

the features of a wiki system such as page revisions, or provide only a basic conflict resolution mechanism 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* [30] (DHT) and *optimistic replication* [31]. At the storage system level, we use DHTs, which have been proved [16] 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, such as WOOT [23] or Logoot [39], which guarantees eventual consistency, causal consistency and intention preservation [35].

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 AOP middleware (Damon http://damon.sf.net). Thus, we were able to reuse existing implementations for all the components needed, and integrate our method transparently.

Section 2 presents approaches related to our proposition and analyzes their strong and weak points. In Section 3, an overview of DHT characteristics and optimistic consistency maintenance algorithms is presented.

The paper further describes, in Section 4, the architecture of the UniWiki system and its algorithms. An implementation of this system is presented in Section 5. In Section 6, a validation via experimentation on a large-scale scenario is demonstrated. The paper concludes in Section 7.

2. Related Work

A lot of systems such as Coral [12], CoDeeN [37] or Globule [24] have been proposed during the last year to address the issue of web hosting on peer-to-peer networks. All these systems belong to the category of Content Delivery Network (CDN). Such systems offers high availability of data stored on peer-to-peer network which is achieved by using simple replication techniques combining with caching mechanisms. The main drawbacks of theses approaches is that they consider only static content while we are interested in storing dynamic content which might be edited concurrently.

One of the most relevant architectural proposals in the field of large-scale collaborative platforms is a semi-decentralized system for hosting wiki web sites like Wikipedia, using a collaborative approach. This design focuses on distributing the pages that compose the wiki across a network of nodes provided by individuals and organizations willing to collaborate in hosting the wiki. The paper [36] presents algorithms for page placement so that the capacity of the nodes is not exceeded and the load is balanced, and algorithms for routing client requests to the appropriate nodes.

In this architecture, only the storage of content wiki pages is fully distributed. Client requests (both read and write) are handled by a subset of *trusted* nodes, and meta-functionalities, such as user pages, searching, access controls and special pages, are still dealt with in a centralized manner.

While this system can resist random node departures and failures (one of the inherent advantages of replication), it does not handle more advanced failures, such as partitioning. Moreover, when concurrent updates of the same content (page) happen, a simple tie breaking rule is used to handle conflicts. Such an approach guarantees consistency but it does not fulfill wiki users expectations (some of the user contributions will be ignored by the system when the tie-breaking policy is applied).

Approaches that relate more to the P2P model include:

- DistriWiki [20], based on the JXTA[TM][14] protocol, provides a P2P wiki architecture, where each

2

node represents a set of wiki pages stored on a user's computer. It concentrates on the communication aspect, but ignores wiki specific features, such as versioning, and does not solve the problem of merging concurrent contributions from different users.

- DTWiki [8] uses delay tolerant network (DTN) [10] as the basis for building a distributed and replicated storage mechanism, enabling offline work and access from terminals with intermittent Internet access.

- Piki [21] adapts Key Based Routing [5] and Distributed Hash Tables [30] (DHTs) to build its storage mechanism, where each node in the P2P network is responsible for storing a number of pages.

- Scalaris [25, 32] proposes the usage of DHTs as the underlying storage mechanism, backed by a safe distributed transaction manager based on Paxos, inheriting all the advantages of DHTs: quasi-reliable distributed storage, fault tolerance, censorship resilience. Scalaris with its transactional approach guarantees strong consistency while our approach with its optimistic replication algorithm ensures a weaker consistency level: *eventual consistency* [31]. Consistency level guaranteed by WOOT (eventual consistency and intention preservation) is more suitable to collaborative applications where the system should never take a decision on his own, when conflicting updates occur, that might lead to ignore some user-generated changes. A more thorough comparison which studies the overheads of the commit-protocol versus the optimistic replication mechanism is out of the scope of this paper, and will be conducted as a future work.

These systems concentrate on the communication and replication aspects, but do not properly address concurrent updates. They either do not consider the problem, or take a transactional approach [32] where only one user can successfully save his changes. Both approaches are clearly not adapted to collaborative authoring since some user's contributions might be lost.

Concurrent editing can be identified as the main concern of other systems:

- RepliWiki [29] aims at providing an efficient way of replicating large scale wikis like Wikipedia, in order to distribute the load and ensure that there is no single point of failure. The central part of the RepliWiki is the Summary Hash History (SHH) [15], a synchronization algorithm that aims at ensuring divergence control, efficient update propagation and tamper-evident update history.

- Wooki [38] is a wiki application built on top of a unstructured P2P network. Propagation of updates among peers is accomplished by the means of a probabilistic epidemic broadcast, and merging of concurrent updates is under control of the WOOT [23] consistency maintenance algorithm. This algorithm ensures convergence of content and intention preservation [35] – no user updates are lost in case of merging.

- Git [13], a distributed version control system (DVCS) where the central source repository is replaced by a completely decentralized P2P network, stands as the underlying storage mechanism for several wiki systems: git-wiki, eWiki, WiGit, Ikiwiki, ymcGitWiki. The advanced merging capabilities of Git make these wiki systems suitable for offline and distributed content authoring. Unfortunately, DVCSs ensure only causal consistency, which means convergence of replicated content is not automatically guaranteed, leaving the conflict resolution to the end users.

Unlike the previous approaches, the latter handle concurrent updates efficiently. However, by employing total replication, where all nodes contain copies of the whole data, they limit their scalability since the number of possible "donors" is drastically reduced by the need to be powerful enough to host the entire wiki system. They are suitable for smaller wikis, and can apply to the scenario of collaborative writing in mobile environments.

3. Background

Distributed Hash Tables (DHT), one of the most well discussed topics regarding P2P systems, provide a simple distributed storage interface, with *decentralization*, *scalability* and *fault tolerance* as its main properties.

Throughout the last decade, several architectures were proposed, each with its various strong points. The most notable are Chord [33], one of the favorites in the academic/research environment, Pastry [30], frequently used in practice, Tapestry [44], CAN [28] and Kademlia [17]. All DHT systems have in common a simple map storage interface, allowing to **put(key, value)** and **get(key)** back values from the storage. Beyond this common ideology, the implementations have critical differences. While some allow storing just one value for each key, others maintain a list of values, which is

updated on each call of **put**. Several implementations even have an expiration mechanism, where a value is associated with a key for a limited period of time, after which it is removed from the list of values for that key.

DHTs have been successfully used in P2P applications, not only in file-sharing programs like BitTorrent [26], but also inside critical systems such as the storage mechanism of the heavily-accessed Amazon [6].

Although DHTs have been popular long before any of the distributed wiki systems have been proposed, few architectures employ them in wiki applications. This is a natural consequence of the fact that only static content can be reliably stored in a DHT, or at most content with infrequent updates originating from one point. Decentralized concurrent updates could create inconsistent states and cause loss of content.

A control mechanism suitable for maintaining consistency of shared data in collaborative editing is provided by the *operational transformation* approach [9, 34]. Instead of sharing the actual document, the peers maintain the list of actions that users performed to create that content. During reconciliation phases, since the linearization of concurrent operations changes the context in which they occur, *operations* must be *transformed* to be relevant in this new context. This linearization is computed in such a way that it ensures convergence of copies regardless of the order in which operations are received. In comparison, traditional consistency algorithms used in version control systems – distributed or not – ensure only causal consistency: all actions are seen in the same order they have been performed, but do not guarantee convergence of copies in presence of concurrent actions.

Moreover, OT approaches ensure an additional correctness criteria called *intention preservation* [35]. Intention preservation guarantee that the effect of operations is preserved while obtaining convergence. User contributions are mixed in order to compute the convergence state rather than being simply discarded.

Any kind of content can be managed using operational transformations, not only linear text, as long as the correct transformations are defined. The difficulty resides in finding these transformations. Several algorithms have been proposed based on this approach, working on environments with different specifics. Unfortunately, most of them require either a central server or the use of vector clocks in order to ensure causality between updates. These requirements limit the potential scalability of OT-based systems.

Another approach is *WOOT*, (WithOut Operational Transformation) [23], based on the same principles as operational transformation, but sacrificing the breadth of the supported content types to gain simplicity. The WOOT algorithm does not require central servers nor vector clocks, and uses a simple algorithm for merging operations. However, it only applies to linear text structures and supports a reduced set of operations: *insert* and *delete*. Basically, the algorithm works[1] as follows.

WOOT sees a document 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*. Formally, a W-character is a five-tuple $< id, \alpha, v, id_{cp}, id_{cn} >$, where:

- *id* is the identifier of the W-character, a pair (ns, ng), consisting of the unique ID of the peer where the block was created, and a logical timestamp local to that peer;

- α is the alphabetical value of the block;

- *v* indicates whether the block is visible or not;

- id_{cp} is the identifier of the previous W-character, after which this block is inserted;

- id_{cn} is the identifier of the following 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

---

[1]For a more detailed description, please refer to [23]

4

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.

WOOT has two major disadvantages: i) it only applies to linear structures, since it supports just two operations, insert and delete, ii) all operations that affected a document since its creation must be preserved, which results in many *tombstones* [23], increasing the required storage space.

The latter disadvantage is avoided in the Logoot algorithm [39], which replaces the relative positioning of new operations with standalone position identifiers. These identifiers can point to the correct target position even after drastic changes of the context on which the operation must be applied, without requiring tombstones. Moreover, the time complexity of applying a new operation in Logoot is only logarithmic in the current document size, compared to n^2 for WOOT, where n is the total number of operations.

The correctness for both the WOOT and Logoot approaches is based on the CCI criteria: Causality, Convergence and Intention preservation [35].

In the following sections, we describe an implementation of UniWiki which is based on the WOOT algorithm. It is worth to mention that the same principles apply for building an implementation based on Logoot.

4. The UniWiki Architecture

The central idea of our paper is to use a DHT system as the storage mechanism, with updates handled by running a consistency algorithm directly in the DHT nodes. This integration is possible by changing the behavior of the **put** and **get** methods. Instead of storing the actual wiki document as plain text content, each DHT node stores the corresponding consistency model (like the W-String for WOOT). On display, the DHT reconstructs a simplified version of the document, which is further processed by a wiki server front-end. On update, the wiki front-end creates a list of changes made by the user, which are transformed into operations that are applied and stored inside the DHT.

Since a continuous synchronization is needed between all sites in a total replicated system, it is practical to restrict their number, as it is done, for instance, in the current architecture of Wikipedia. However, as DHTs allow accesses from many clients at the same time, in our approach wiki front-ends can be placed in numerous sites, without the need for any synchronization between them. Front-ends can be added or removed at any

moment, because the entire data is stored outside the front-end, in the DHT. Also, the specifics of DHT systems allow adding and removing peers from the DHT even at high rates, thus opening the storage system for third-party contributions.

4.1. Overview

Basically, our system consists of two parts, as depicted in Figure 1: the DHT network, responsible for data storage, and the 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 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. The information is stored on a peer in the form of operations, conforming to the data model of the consistency algorithm chosen. 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 and Logoot algorithms ensure eventual consistency – convergence – on the replicated content stored by peers responsible for the same keys. In the implementation we have chosen FreePastry [30]

The **wiki front-ends** are responsible for handling client requests, by retrieving the data 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 operations, then sends them to the DHT to be integrated, stored and propagated to the replicated peers.

4.2. Data model

As explained in section 3 and in [23], consistency algorithms work not with the plain textual content of a document, but with an enriched version of it, like the W-string of WOOT. This is the **page model** that is stored inside the DHT, at each peer responsible for that document.

To update this model, **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,
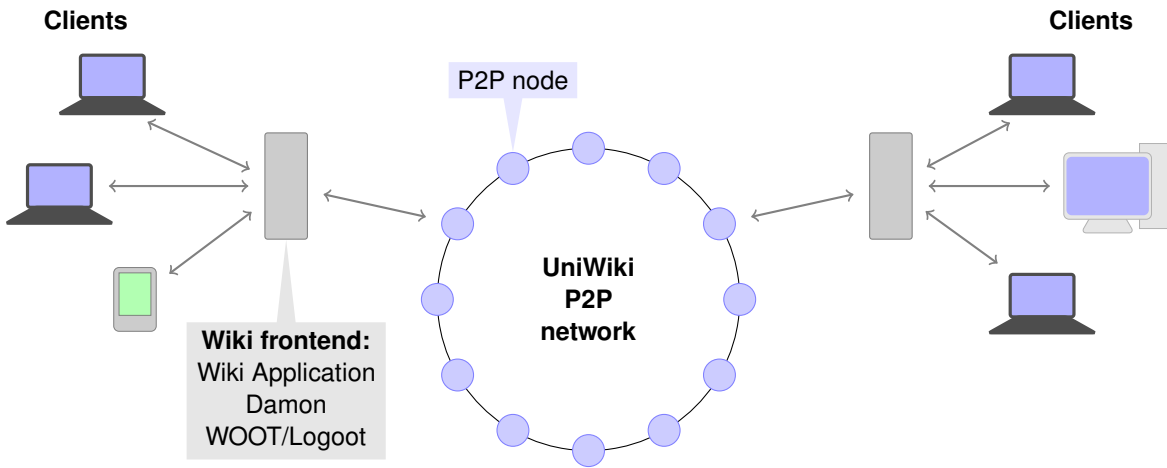
5

Figure 1: UniWiki architecture overview.

but also the original version on which he started editing. And, since transforming textual changes into operations requires knowing the 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 page model is needed, but just enough to be able to determine the context in which changes occur. For WOOT, only the visible part of the linearized W-string is needed, and only the *id* and actual block content. In the case of Logoot, the whole model contains just the needed information, the line content and its identifier. In general, this simplified model – or, from a different point of view, enriched content – is called the **front-end 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. This is the actual content that the wiki engine uses.

4.3. Algorithms

This section describes in term of algorithms the behaviors of a front-end server and of a DHT peer.

4.3.1. 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 front-end page from the DHT, using the hashed name of the requested wiki content as the key. The received page is transformed into plain wiki content, which is rendered as HTML if necessary, then sent to the client.

---

onDisplay(contentURI)
   pageName = getTargetPage(contentURI)
   frontendPage = dht.get(getHash(pageName))
   wikiContent = extractContent(frontendPage)
   htmlContent = renderHTML(wikiContent)
   return htmlContent

For example, let's say that the client requested `http://host.net/Main/Home`, a wiki page named `Main/Home`, whose content is:

```
A wiki is an easy-to-edit //website//
that will help you work better together.
You're currently in the **Main** space,
looking at its **Home** page.
```

From the URI, the wiki engine determines the name of the requested page (`Main/Home`), and proceeds to load the content of the page from the storage. UniWiki intercepts this call, and instead retrieves the content from the distributed storage, as a front-end page, and eliminates all the meta-data from it to obtain the plain content. Supposing the hash of this page name is `c7ad44cbad762a5da0a452f9e854fdc1`, this is the key which is requested from the DHT. The returned fronted page corresponding to a WOOT model would be:

```
{(N1, 1), "A wiki is an easy-to-edit //website//"}
{(N1, 2), "that will help you work better together."}
{(N2, 1), "You're currently in the **Main** space,"}
{(N2, 2), "looking at its **Home** page."}
```

6

This indicates that the content has been changed on two peers, N1 and N2, each one adding one phrase, which, given the low sequence numbers, are probably unedited.

Extracting the wiki content is as simple as keeping only the content of the line, without the ID. Then, rendering the wiki markup, which means replacing `**` by bold and `//` by italic, is the job of the wiki engine, and which is of no interest to our paper.

Similarly, 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 (the *intention* of the user is 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)
   pageName = getTargetPage(contentURI)
   frontendPage = dht.get(getHash(pageName))
   session.store(pageName, frontendPage)
   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 [22] is used to compute modifications between this old version and the new version submitted by the client. These modifications are then mapped[2] on the old version of the page model in order to generate operations. Finally, these operations are submitted to the DHT.

---

onSave(contentURI, newWikiContent)
   pageName = getTargetPage(contentURI)
   oldPage = session.get(pageName)
   oldWikiContent = extractContent(oldPage)
   changes[] = diff(oldWikiContent, newWikiContent)
   operations[] = changes2Ops(changes[], oldPage)
   dht.put(getHash(pageName), operations[])

In our example page, suppose another user, logged on the second host, thinks that `together` does not properly fit together with the ambiguous `you`, and decides to update the text as:

---

[2]A thorough description of the mapping algorithm is provided in [38].

```
A wiki is an easy-to-edit //website//
that will help your team work better together.
You're currently in the **Main** space,
looking at its **Home** page.
```

This is the **newWikiContent** that is passed to the method. Since the old content was preserved in the server session, the differences can be computed, and the algorithm determines that the changes are:

- the old line "`that will help you work better together.`" was removed from the second position

- the new line "`that will help your team work better together.`" was inserted between the first and third line from the initial content

Converting these changes to their equivalent WOOT operations is simple, resulting in:

```
del((N1, 2))
ins((N1, 1), (N2, 3), "that will ...", (N2, 1))
```

These two operations are then pushed to the DHT nodes responsible for hosting the key which corresponds to the `Main/Home` page, `c7ad44....`

Since the consistency algorithm running on the DHT is responsible for handling concurrent edits, the front-end does not need to perform any additional actions, like checking for locks or comparing the version on which the user started editing with the current version. Simply pushing these operations, which reflect the user's intention, to the consistency algorithm is enough to assure that the user's changes will be reflected in the wiki page, along with other concurrent changes.

4.3.2. 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 intended behaviors differ from the basic behaviors provided by any DHT since the type of the value returned by a get request – a front-end page – is not the same as the type of the value stored by a put request – a list of operations –, and different from the internal data actually stored in the node – a page model. Furthermore, when replicas of the same page are synchronized, the final value is obtained as the reunion of all the operations from both replicas, and not as a clone of the value from one of the peers, as is the case in a default DHT synchronization.

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 page model corresponding to the key is retrieved from the local storage, the simplified front-end page is extracted, and then sent back to the requester – a front-end server.

```
onGet(key)
    pageModel = store.get(key)
    frontendPage = extractFrontendPage(pageModel)
    return frontendPage
```

The WOOT model corresponding to the example page, as it was before the update, would be:

```
((N1, 1), "A wiki is an easy-to-edit...", T, (S),     (E))
((N1, 2), "that will help you...",        T, (N1, 1), (E))
((N2, 1), "You're currently in the..",    T, (S),     (E))
((N2, 2), "looking at its **Home** page.", T, (N2, 1), (E))
```

That is, four insert operations, where (S) and (E) are special position identifiers corresponding to the start, respectively end of the content. It can be seen that the two phrases were introduced in parallel, independently by the two hosts.

Extracting the front-end page involves linearizing the content, which, in this example, results in putting the lines from N2 after the lines from N1, hiding the deleted lines (none in this case), and keeping only the identifier and the content of each line, since the predecessor and successor identifiers are not needed. The resulting front-end page was listed above.

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

```
onPut(key, ops[])
    pageModel = store.get(key)
    for (op in ops[])
        integrate(op, pageModel)
        pageModel.log(op)
```

When updating the example document, the two operations can be applied directly, since the preconditions required by WOOT are met – the existence of the deleted line and of the context lines around the insert. Thus, the new model becomes:

```
((N1, 1), "A wiki is an easy-to-edit...", T, (S),     (E))
((N1, 2), "that will help you...",        F, (N1, 1), (E))
((N2, 3), "that will help your...",       T, (N1, 1), (N2, 1))
((N2, 1), "You're currently in the..",    T, (S),     (E))
((N2, 2), "looking at its **Home** page.", T, (N2, 1), (E))
```

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.

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

```
onRecover(key)
    pageModel = store.get(key)
    ops[] = pageModel.getLog()
    digests[] = digest(ops[])
    missingOps[] = getReplica(key).antiEntropy(digests[])
    for (op in missingOps[])
        integrate(op, pageModel)
        pageModel.log(op)

onAntiEntropy(key, digests[])
    pageModel = store.get(key)
    ops[] = pageModel.getLog()
    mydigests[] = digest(ops[])
    return notin(ops[], mydigests[], digests[])
```

5. Implementation

Wikis are currently 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., 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 4.3.1. 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 networks. For these reasons, we need a middleware platform that provides the necessary abstractions and mechanisms to construct distributed applications.

In this work, 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 powerful distributed interception techniques (i.e., distributed AOP). 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 AOP middleware Damon [19]. Using this middleware, developers can implement and compose distributed aspects in large-scale environments. Such distributed aspects, activated by local point-cuts (source hooks), trigger remote calls via P2P routing abstractions. A more detailed description of Damon middleware services is presented in [18].

5.1. Damon Overview

The separation of concerns principle addresses a problem where a number of concerns should be identified and completely separated. AOP is an emerging paradigm that presents the principle of separating crosscutting concerns, allowing less interdependence, and more transparency. In AOP, interception is performed in a *join point*, a point in the execution flow, and defined inside a *point-cut*, a set of join points. Whenever the application execution reaches one point-cut, an *advice* associated with it, namely a callback, is executed. This allows the addition of new behaviors with a fully separation of concerns, where developers compose different aspects into a complete application.

However, AOP has not been successfully applied to developing distributed concerns yet (e.g., distribution). In this setting, new abstractions and mechanisms are needed to implement these distributed concerns. Indeed, distributed AOP presents a different philosophy than traditional solutions like remote object or component models. When developers design a new application, they firstly obtain or implement the *raw* application without any distributed concerns in mind. They may simply design the main concern by *thinking in local*, and later implementing the rest of the distributed concerns, designing the necessary connectors (e.g., remote point-cuts), which conform the distributed AOP application.
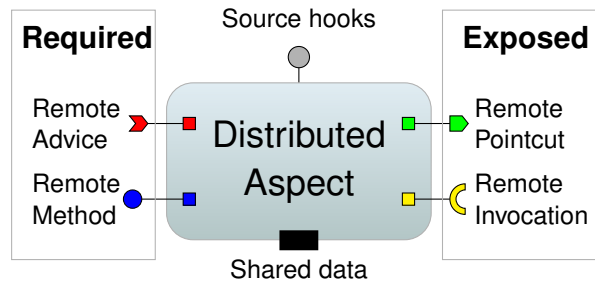

Figure 2: The structure of a Distributed Aspect.

By making efficient use of DHT substrates and dynamic interception mechanisms, Damon is designed as a distributed AOP middleware based on P2P and dynamic AOP mechanisms. In this setting, Damon provides new non-functional properties (e.g., scalability) to existent or new applications or systems transparently. In order to achieve this goal it offers the necessary abstractions and services to implement distributed concerns in the form of distributed aspects for large-scale environments.

The main features of Damon are:

- **P2P Abstractions**: taking advantage of the built-in DHT-based P2P network. This middleware allows key-based, direct-node, any-node, many-nodes, neighbor-nodes, or all-nodes routing and group abstractions.

- **Scalable Deployment Platform**: this layer facilitates the deployment and activation of distributed aspects in large-scale scenarios. Moreover, messaging, persistence, and reflection services are offered for this layer.

- **Distributed Composition Model**: this model allows connection-oriented and meta-level features. In addition, the composition model allow us to

make a good abstraction, clear interaction scheme, and reconfiguration in runtime.

The main components of a distributed aspect are depicted in figure 2 and summarized below:

- The **source hooks** are the locally executed methods that trigger the aspect. Similar to classical AOP point-cuts, they are the contract that indicate when the distributed aspect should be executed.

- The **remote point-cuts** is a service for remotely propagating associated join points. Distributed aspects can disseminate point-cuts to a single host or to a group of hosts. Point-cuts are identified using a simple name (String), which allows distributed aspects to work together without being tightly linked.

- The **remote advices** are the remote services triggered by the execution of a remote point-cut. By using the same identifier as an existing remote point-cut, the connection between the point-cut and its associated remote advices is transparent, without any declared code dependencies or remote method invocations. This transparency allows Damon to separate and to dynamically add distributed concerns to legacy code, without requiring any changes.

- The **remote methods/invocations** are meant to allow inter-aspect communication on demand. Using this mechanism, data can be sent back from the remote aspects to the origin point-cut.

- **Shared data** allows stateful distributed aspects to save/restore their state information. Moreover, in group aspects, this state can be shared by one, many or all members of the group.

The Damon middleware is presented in detail in [18, 19].

Following these ideas, 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.

5.2. UniWiki Implementation

Traditional wiki applications are executed locally on the wiki front-end. This scenario is ideal for applying distributed interception, because we can intercept the local behavior to inject our algorithms. Thereby, using Damon, we can model transparently the necessary concerns:

- *Distribution*: refers to dissemination and storage of wiki pages into the system. This dissemination allows a load-balanced distribution, where each node contains a similar number of stored wiki pages. In addition, this concern also guarantees that clients can access always to data in a single and uniform way.

- *Replication*: data is not only stored on the responsible node, since it would become unavailable if this node fails, or leaves the network. Thus, this concern allows to these nodes to copy their wiki pages in other nodes. Moreover, they have to maintain these copies, in order to guarantee a specific number of alive replicas at any moment.

- *Consistency*: when multiple clients are saving and reading concurrently the same wiki pages, data can become inconsistent. This approach proposes to generate operation logs (i.e., patches) of the wiki pages and distribute them. Finally, the final wiki page is regenerated from stored patches.

Figure 3 presents the UniWiki source hook, where we aim to intercept locally 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, aspect 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 aspects.

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

We now describe the UniWiki execution step by step as shown in Figure 4, 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.

Later, the replication concern is also based on P2P mechanisms, following a neighbor replication strategy [16]. Two distributed meta-aspects are used to implement this concern in the back-end: the **Replicator** (intercepting the Storage) and the **ReplicaStore** instances (many per Replicator).

Finally, as explained in the previous section, the consistency concern is centered on the deployed consistency algorithm (i.e., WOOT). In this implementation, it allows edition, patching, and merging of wiki pages, and it performs these operations via distribution concern calls interception. Again, two distributed meta-aspects
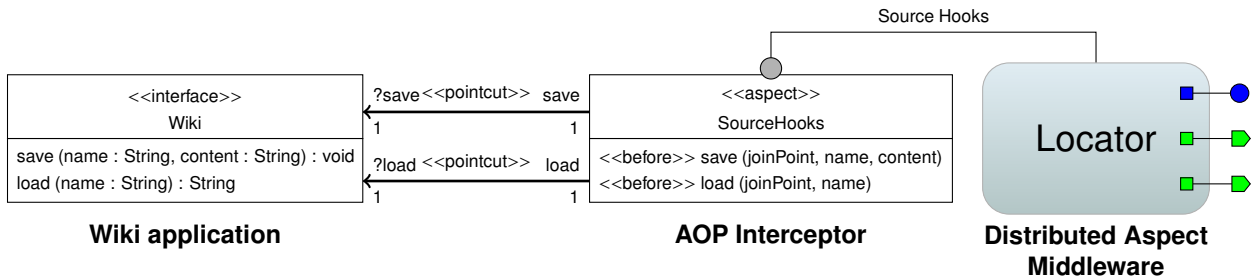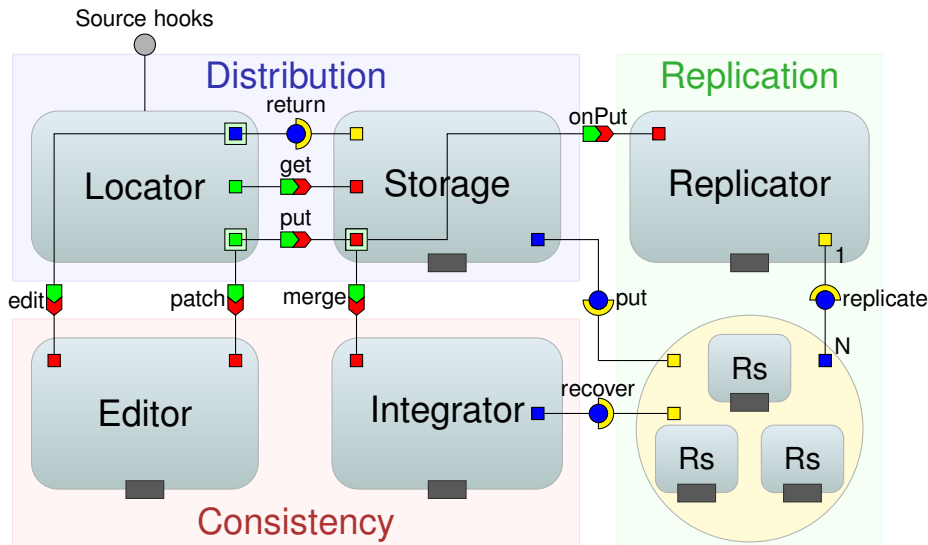
Figure 3: Wiki Source Hook Interface.

Figure 4: UniWiki distributed aspects and their interaction.

are used to implement this concern: the **Editor** (front-end) intercepting the Locator, and the **Integrator** (back-end), intercepting the Storage, and interacting with the ReplicaStore.

Distribution:

1. 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. Afterwards, the Locator distributed aspect 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.

2. These save method executions are propagated using the **put** remote point-cut. Consequently, the remote point-cuts are routed to the key owner node, by using their URL to generate the necessary key.

3. Once the key has reached its destination, the registered connectors are triggered on the Storage in-

stance 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.

4. Finally, it propagates an asynchronous response using the **return** call via direct node routing. 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 meta-aspects in order to extend or modify the current application behavior in runtime. Thereby, thanks to the meta-level approach, we are able to change active concerns (e.g., new policies), or reconfiguring the system in order to adapt it.

Replication:

1. When dealing with the save method case, we need

11

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, since it would surely become unavailable if this host leaves the network for any reason. In order to address this problem, we activate the Replicator distributed meta-aspect in runtime, which follows a specific policy (e.g., neighbor selection strategy [16]). The Replicator has a remote meta-point-cut called **onPut**, which intercepts the Storage put requests from the Locator service in a transparent way.

2. 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.

3. Finally, ReplicaStore distributed meta-aspects 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 wiki page, using a remote meta-advice **put**, in order to replace the Locator remote point-cut.

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 meta-aspects, which intercept the DHT-based calls to perform the consistency behavior. Their task is the modification of the distribution behavior, by adding the patch transformation in the edition phase, and the patch merging in the storage phase.

1. The Editor distributed meta-aspect owns a remote meta-point-cut (**edit**) that intercepts the return remote invocations from Storage to Locator instances. This mechanism stores the original value in its own session data. Obviously, in a similar way, the Integrator prepares the received information to be rendered as a wiki page.

2. Later, if the page is modified, a save method triggers the put mechanism, where another remote point-cut (**patch**) transforms the wiki page into the patch information, by using the saved session value.

3. 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 replacing the wiki page with consistent patch information.

4. In this setting, having multiple replicated copies leads to inconsistencies. We use the *antiEntropy*

technique [7], in order to recover a log of differences among each page and its respective replicas. Using the **recover** remote invocation, the Integrator sends the necessary patches to be sure that all copies are consistent.
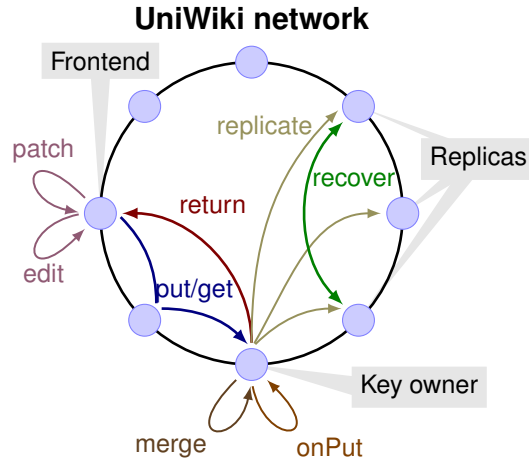

Figure 5: Peer interaction.

Finally, we summarize the UniWiki connections (network scheme in Figure 5) among the distributed aspects and meta-aspects:

- Distribution:

  - The **put** and **get** point-cuts are forwarded to the host responsible for the value associated with the current key, using the *DHT-based routing*.

  - The **return** of the requested value from the key owner to the front-end is done as a *direct* abstraction.

- Consistency:

  - The **patch** and **edit** meta-point-cuts of the Editor aspect are executed *locally*, on the front-end serving a wiki request.

  - Upon receiving a patch, the **merge** meta-point-cut of the Integrator aspect is executed *locally* on the node responsible for the key.

- Replication:

  - At the same moment, the **onPut** meta-point-cut of the Replicator aspect is executed *locally* on the node responsible for the key, and:

– Forwards the new content using the **replicate** remote invocation as a *direct* call to all the *neighbors* responsible for the same key.

– When a new host joins the network, or recovers from a problem, the Integrator aspect running on it will re-synchronize with a randomly selected replica, using the **recover** remote invocation as a *direct* call.

5.3. Technical implementation details

While the framework is flexible enough to allow any of the components to be changed, in the current implementation, which was also used for the experimentation phase, we settled on the following components:

- the AspectWerkz AOP framework for local interception, i.e. for inserting UniWiki into existing wiki engines

- FreePastry as the DHT/KBR implementation, for its efficient Java implementation; in addition, another advantage of FreePastry is that it provides network proximity aware routing, which helps maintain a low communication time between the front-end and the DHT

- WOOT as the consistency algorithm

- we built a very simple wiki engine as a testing platform, which originally stores the wiki pages in a `Map`

6. UniWiki Evaluation

We have conducted several experiments to measure the viability of our UniWiki system. For this, we have used Grid'5000 [4] platform, a large-scale distributed environment that can be easily controlled, reconfigured and monitored. The platform disposes of 5000 CPUs distributed over 9 sites in France. Every site hosts a cluster and all sites are connected by a high speed network.

More precisely, our experiments ran on 120 real nodes from the Grid'5000 network, from different geographical locations in France, including Nancy, Rennes, Orsay, Toulouse, and Lille.

6.1. Testbed

The analysis of our large scale system focuses on three main aspects: load-balancing in data dispersion among all nodes (distribution), failed hosts that do not
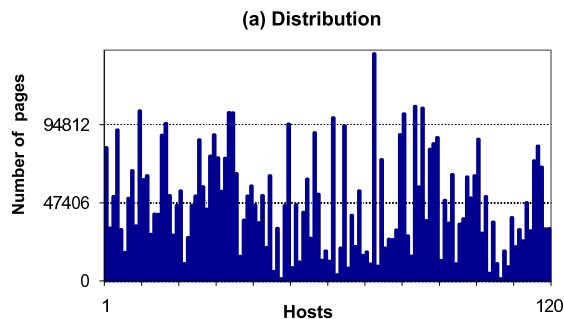

Figure 6: Empirical results - Distribution.

affect system behavior (replication), and the operation performance (consistency).

Regarding the distribution, we study the data dispersion in the network.

- *Objective*: To demonstrate that when our system works with a high number of hosts, is able to store a real large data set of wiki pages, and that all the information is uniformly distributed among them.

- *Description*: We create a network of 120 hosts, and, using a recent Wikipedia snapshot, we introduce their 5,688,666 entries. The idea is that data distribution is uniform, and each host has a similar quantity of values (wiki pages).

- *Results*: We can see in Figures 6 that we have a system working and the results are as expected. Thereby, the distribution of data trends to be uniform. Results indicate that each host has an average of 47,406 stored wiki pages, and using an approximate average of space (4.24 KB) per wiki pages we have 196 MB, with a maximum of 570.5 MB (137,776 values) in one node.

- *Why*: Uniform distribution of data is guaranteed by the key-based routing substrate, and by the hash function (SHA-1) employed to generate the keys. However, with this number of hosts (120) the distribution values show that a 54.17% are over the average, and a 37.5% are over the double of the average. Furthermore, we can see similar results in simulations, with a random distribution of 120 nodes. For these simulations we have used Planet-Sim [27], which is an object oriented simulation framework for overlay networks.

We make another simulation with 1000 nodes using the same DHT simulator. The results of this

last simulation (Figure 7) shows that the values are: 63.7% over average and 20.6% over double of the average, because with a high number of nodes the uniformity is improved.
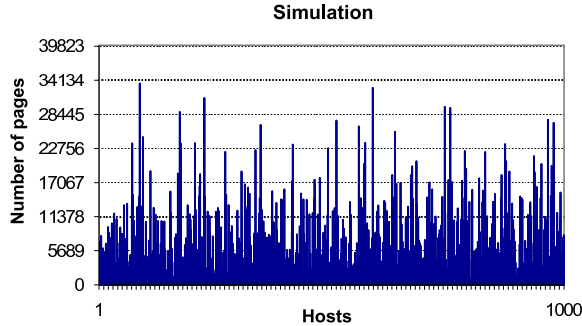

Figure 7: Wikipedia data distribution.

Secondly, we study the fault-tolerance of our system platform.

- *Objective*: To demonstrate that in a real network with failures, our system continues working as expected.

- *Description*: In this experiment we introduce problems on a specific fraction of the network. In this case, each host inserts 1000 wiki pages, and retrieves 1000 on them randomly. The objective of this test is to check persistence and reliable properties of our system. After the insertions, a fraction of the hosts fail without warning, and we try to restore these wiki pages from a random existing host.

- *Results*: We can see the excellent results in Figure 8. Even in the worst case (50% of network fails at the same time), we have a high probability (average of 99%) to activate the previously inserted wiki pages.

- *Why*: The theoretical probability that all the replicas for a given document fail is:

$$\prod_{i=1}^{n} r_i = r_1 \cdot r_2 \cdot ... \cdot r_n = (r)^n$$

where n is the replication factor and r the probability that a replica fails. For instance, when r = 0.5, and n = 6, the result is $(0.5)^6 \approx 0.015 \approx 1.5\%$.

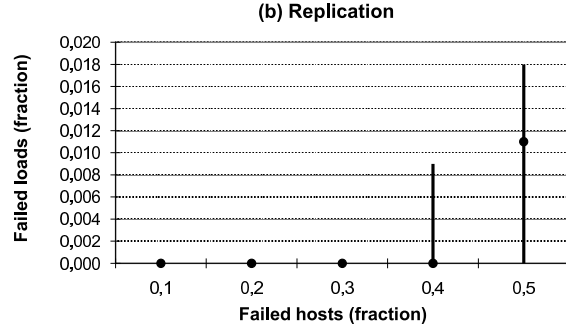Finally, for consistency we study the performance of our operations.

Figure 8: Empirical results - Replication.

- *Objective*: To demonstrate that our routing abstraction is efficient in terms of time and network hops.

- *Description*: Similarly to the previous experiment, we create a 120 hosts network, and each host inserts and retrieves 1000 times the same wiki page. The idea is that the initial value is modified, and retrieved concurrently. The experiment is repeated twice: the first time with consistency mechanisms disabled (only the put call), and the second time with these mechanisms enabled (patch and merge). In the last step, the consistency of the wiki page is evaluated for each host.

- *Results*: For this experiment, we found that the consistency is guaranteed by our system. We can see the operation times in Figure 9, and for each operation the number of hops has an average of 2. Therefore, the wiki page put operation has an average time of **145** ms, and an overall overhead of: **3.45**. For update operations the value is logically higher : **164** ms. with an overhead of **3.90**. Finally, the update operation overhead respects put operation, when the consistency operation is performed, is **1.13**.

- *Why*: Due to the nature of the Grid experimentation platform, latencies are low. Moreover, we consider that the operation overhead is also low. Finally, theoretical number of hops is logarithmic respect the size of the network. In this case, log (120 hosts) = 2 hops.

7. Conclusions and Future Work

In this article we propose an efficient P2P system for storing distributed wikis, extended to large-scale scenar-
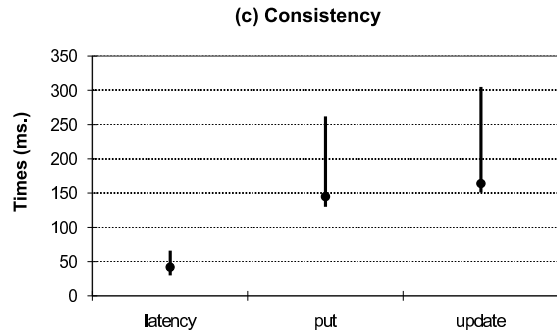
(c) Consistency

Figure 9: Empirical results - Consistency.

ios transparently. In this line we present the algorithms, a prototype, and the experimentation.

Nowadays, many solutions for wiki distribution are proposed, but as we can see in the Section 2, the current approaches focus on only one half of the problems, failing to address both efficient distribution and correct replication. 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.

Validation of UniWiki has been conducted on the Grid'5000 testbed. We have proved that our solution is viable in large-scale scenarios, and that the system has acceptable performance. Our experiments where conducted with real data [41] from Wikipedia which include almost 6 million entries.

The initial prototype, freely available at `http://uniwiki.sf.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.

8. Acknowledgments

References

[1] L. Allen, G. Fernandez, K. Kane, D. Leblang, D. Minard, and J. Posner. ClearCase MultiSite: Supporting Geographically-Distributed Software Development. In J. Estublier, editor, *Software Configuration Management: Selected Papers of the ICSE SCM-4 and SCM-5 Workshops*, number 1005 in Lecture Notes in Computer Science, pages 194–214. Springer-Verlag, Oct. 1995.

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

[3] M. Bergsma. Wikimedia architecture. `http://www.nedworks.org/~mark/presentations/san/Wikimedia%20architecture.pdf`, 2007.

[4] R. Bolze, F. Cappello, E. Caron, M. Dayd, F. Desprez, E. Jeannot, Y. Jgou, S. Lantri, J. Leduc, N. Melab, G. Mornet, R. Namyst, P. Primet, B. Quetier, O. Richard, E.-G. Talbi, and I. Touche. Grid'5000: A Large Scale and Highly Reconfigurable Experimental Grid Testbed. *International Journal of High Performance Computing Applications*, 20(4):481–494, Nov. 2006.

[5] F. Dabek, B. Zhao, P. Druschel, J. Kubiatowicz, and I. Stoica. Towards a Common API for Structured Peer-to-Peer Overlays. *Lectures Notes in Computer Science*, 2735:33–44, 2003.

[6] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's Highly Available Key-Value Store. In *Proceedings of 21st ACM SIGOPS symposium on Operating systems principles - SOSP 2007*, pages 205–220. ACM Press, 2007.

[7] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of the 6th annual ACM Symposium on Principles of distributed computing - PODC 87*, pages 1–12, New York, NY, USA, 1987. ACM.

[8] 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*, pages 945–952, New York, NY, USA, 2008. ACM Press.

[9] C. A. Ellis and S. J. Gibbs. Concurrency control in groupware systems. In *SIGMOD Conference*, volume 18, pages 399–407, 1989.

[10] K. Fall. A Delay-tolerant Network Architecture for Challenged Internets. In *Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communications - SIGCOMM 2003*, pages 27–34. ACM Press, 2003.

15

[11] W. Foundation. 2008-2009 annual report. `http://upload.wikimedia.org/wikipedia/foundation/a/a4/WMF_Annual_Report_20082009_online.pdf`, 2009.

[12] M. J. Freedman, E. Freudenthal, and D. Mazières. Democratizing content publication with coral. In *Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation - NSDI 2004*, pages 18–18, Berkeley, CA, USA, 2004. USENIX Association.

[13] Git – fast version control system. `http://git.or.cz/`.

[14] L. Gong. JXTA: A Network Programming Environment. *IEEE Internet Computing*, 5(3):88–95, 2001.

[15] B. Kang. Summary Hash History for Optimistic Replication of Wikipedia. `http://isr.uncc.edu/shh/`.

[16] 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*, pages 90–97, Oulu, Finland, Dec. 2007. ACM Press.

[17] P. Maymounkov and D. Mazières. Kademlia: A Peer-to-Peer Information System based on the XOR Metric. *Lecture Notes In Computer Science*, 2429:53–65, 2002.

[18] R. Mondéjar, P. García, C. Pairot, and A. Skarmeta. Building a distributed AOP middleware for large scale systems. In *Proceedings of the workshop on Next generation aspect oriented middleware - NAOMI 2008*, pages 17–22, Brussels, Belgium, 2008. ACM Press.

[19] 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*, pages 539–540, Honolulu, Hawaii, USA, Mar. 2009. ACM Press.

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

[21] P. Mukherjee, C. Leng, and A. Schürr. Piki - A Peer-to-Peer based Wiki Engine. In *Proceedings of the IEEE International Conference on Peer-to-Peer Computing - P2P 2008*, pages 185–186, Los Alamitos, CA, USA, 2008. IEEE Computer Society.

[22] E. W. Myers. An O(ND) Difference Algorithm and its Variations. *Algorithmica*, 1(2):251–266, 1986.

[23] 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*, pages 259–267, Banff, Alberta, Canada, Nov. 2006. ACM Press.

[24] G. Pierre and M. van Steen. Globule: a collaborative content delivery network. *IEEE Communications Magazine*, 44(8):127–133, Aug. 2006. `http://www.globule.org/publi/GCCDN_commag2006.html`.

[25] S. Plantikow, A. Reinefeld, and F. Schintke. Transactions for Distributed Wikis on Structured Overlays. In *Proceedings of the 18th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management - DSOM 2007*, volume 4785, pages 256–267. Springer-Verlag, Oct. 2007.

[26] J. Pouwelse, P. Garbacki, D. Epema, and H. Sips. The Bittorrent P2P File-Sharing System: Measurements and Analysis. *Lecture Notes in Computer Science*, 3640:205, 2005.

[27] J. Pujol-Ahulló and P. García-López. Planetsim: An extensible simulation tool for peer-to-peer networks and services. In *Proceedings of 9th International Conference on Peer-to-Peer Computing - P2P 2009*, pages 85–86, Piscataway, NJ08855-1331, 2009. IEEE Computer Society.

[28] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker. A Scalable Content-addressable Network. In *Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communications - SIGCOMM 2001*,

[29] RepliWiki – A Next Generation Architecture for Wikipedia. `http://isr.uncc.edu/repliwiki/`.

[30] A. Rowstron and P. Druschel. Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms - Middleware 2001*, volume 2218, pages 329–350. Springer-Verlag, Nov. 2001.

[31] Y. Saito and M. Shapiro. Optimistic Replication. *ACM Computing Surveys*, 37(1):42–81, 2005.

[32] T. Schütt, F. Schintke, and A. Reinefeld. Scalaris: Reliable transactional P2P key/value store. In *Proceedings of the 7th ACM SIGPLAN Workshop on Erlang - Erlang'08*, pages 41–48. ACM Press, 2008.

[33] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Protocol for Internet Applications. *IEEE/ACM Transactions on Networking*, 11(1):17–32, 2003.

[34] C. Sun and C. A. Ellis. Operational transformation in real-time group editors: Issues, algorithms, and achievements. In *Proceedings of the ACM Conference on Computer Supported Cooperative Work - CSCW '98*, pages 59–68, New York, NY, USA, Nov. 1998. ACM Press.

[35] C. Sun, X. Jia, Y. Zhang, Y. Yang, and D. Chen. Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems. *ACM Transactions on Computer-Human Interaction*, 5(1):63–108, Mar. 1998.

[36] G. Urdaneta, G. Pierre, and M. van Steen. A decentralized wiki engine for collaborative wikipedia hosting. In *Proceedings of the 3rd International Conference on Web Information Systems and Technologies, WEBIST 2007*, Mar. 2007.

[37] L. Wang, K. S. Park, R. Pang, V. Pai, and L. Peterson. Reliability and Security in the CoDeeN Content Distribution Network. In *Proceedings of the annual conference on USENIX Annual Technical Conference - ATEC 2004*, pages 14–14, Berkeley, CA, USA, 2004. USENIX Association.

[38] S. Weiss, P. Urso, and P. Molli. Wooki: a P2P Wiki-based Collaborative Writing Tool. *Lecture Notes In Computer Science*, 4831(1005):503–512, Dec. 2007.

[39] S. Weiss, P. Urso, and P. Molli. Logoot : A Scalable Optimistic Replication Algorithm for Collaborative Editing on P2P Networks. In *Proceedings of the 29th International Conference on Distributed Computing Systems - ICDCS 2009*, pages 404–412. IEEE Computer Society, June 2009.

[40] Wikipedia, the online encyclopedia. `http://www.wikipedia.org/`.

[41] Wikipedia Data, 2008. `http://download.wikimedia.org/enwiki/latest/`.

[42] Wikipedia Statistics, 2008. `http://meta.wikimedia.org/wiki/Statistics`.

[43] Code Co-op. `http://www.relisoft.com/co_op/`.

[44] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Routing. Technical report, University of California at Berkeley, Berkeley, CA, USA, 2001.

pages 161–172. ACM Press, 2001.