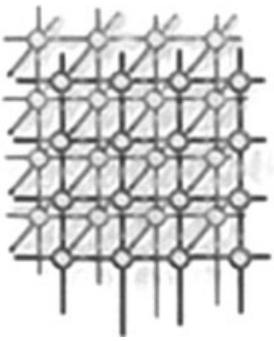
CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2010; **22**:831–871
Published online 23 July 2009 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe.1465



A flexible content repository to enable a peer-to-peer-based wiki

Udo Bartlang[1,*,†] and Jörg P. Müller[2]

[1]*Siemens AG, Corporate Technology, Information and Communications, Otto-Hahn-Ring 6, D-81739 Munich, Germany*
[2]*Clausthal University of Technology, Department of Informatics, Julius-Albert-Str.4, D-38678 Clausthal-Zellerfeld, Germany*

SUMMARY

Wikis—being major applications of the Web 2.0—are used for a large number of purposes, such as encyclopedias, project documentation, and coordination, both in open communities and in enterprises. At the application level, users are targeted as both consumers and producers of dynamic content. Yet, this kind of peer-to-peer (P2P) principle is not used at the technical level being still dominated by traditional client–server architectures. What lacks is a generic platform that combines the scalability of the P2P approach with, for example, a wiki's requirements for consistent content management in a highly concurrent environment. This paper presents a flexible content repository system that is intended to close the gap by using a hybrid P2P overlay to support scalable, fault-tolerant, consistent, and efficient data operations for the dynamic content of wikis. On the one hand, this paper introduces the generic, overall architecture of the content repository. On the other hand, it describes the major building blocks to enable P2P data management at the system's persistent storage layer, and how these may be used to implement a P2P-based wiki application: (i) a P2P back-end administrates a wiki's actual content resources. (ii) On top, P2P service groups act as indexing groups to implement a wiki's search index. Copyright © 2009 John Wiley & Sons, Ltd.

Received 15 January 2009; Accepted 29 April 2009

KEY WORDS: P2P; content repository; wiki; web 2.0; web 3.0

INTRODUCTION

Gartner [1] among others has observed converging key trends that drive the need for *distributed* management of content. For example, the increase in working over the Internet and the distributed

*Correspondence to: Udo Bartlang, Siemens AG, Corporate Technology, Information and Communications, Otto-Hahn-Ring 6, D-81739 Munich, Germany.
†E-mail: research@bartlang.de, udo.bartlang.ext@siemens.com

collaboration within enterprises requires the sharing of produced data. But especially the explosion of unstructured content data that complicates filtering, administration, and controlled exchange.

For example, intra-enterprise knowledge management aims to facilitate and optimize the retrieval, transfer, and storage of knowledge content. However, the sole exchange of such content is difficult: inconsistencies between redundant content may lead to problems and additional efforts [2]. The common practice in enterprises to employ various storage locations, for instance, an employee's local workstation, group storage devices, or intranet servers demands for knowledge content consolidation.

The latest developments [3] recommend the usage of specialized *content repositories* to enable the management of both structured and unstructured content. Typically, these systems act as a meta layer on top of traditional persistent data stores, such as database management systems, providing additional capabilities. Regarding the design and implementation, however, a state-of-the-art approach of a content repository is primarily based on a centralized architecture.

For instance, distributed database systems as an example for hierarchical client–server systems may split large content data sets to different physically distributed network nodes to establish more efficient data querying through parallelism [4]. However, if replication strategies are applied in distributed systems, the consistency of data needs to be ensured. Therefore, these techniques usually employ a point of *central* coordination. Such *flat* client–server architectures are well suited for static networks and computing infrastructures, where the need for hardware resources can be predetermined quite well. Considering, however, the availability of crucial content, if the single server fails, the whole system service is no longer available, which is known as a the single point of failure.

In contrast, the *peer-to-peer* (P2P) paradigm offers a more *flexible* communication pattern migrating to more and more application domains. For instance, there has been a significant increase of P2P-based systems regarding their popularity and their employment for content distribution on the Internet [5]. The increase in storage capacities, processor power of commodity hardware, and technological improvements to network bandwidth—accompanied by the reduction of its costs—foster decentralized solutions by pushing computer power to the edge of networks. For instance, today even commodity desktop machines are able to store huge amounts of content data and to act as the basis for building sophisticated computing infrastructures [6].

Employing dedicated content repositories is a change in the perspective of content life cycle management [7]. Even with evolving efforts to facilitate this shift of content management perspective, however, today's content repositories are less *flexible* regarding the support of different content models, offered functionality as dynamic runtime reconfiguration, or distributed system models. For example, despite the cognition to distinguish between different types of content, explicitly known semantic of content data (as the degree of importance) is neglected. But semantics of such knowledge regarding certain content types may be exploited, for instance, to optimize the overall system performance supporting a policy-based approach.

This paper presents the method of using a flexible content repository system to implement a P2P-based wiki engine to achieve a more decentralized vision of a dynamic environment for the future Web 3.0. Wikis [8] are popular applications of the so-called Web 2.0 [9], for example, the WWW-based collaborative encyclopedia Wikipedia [10] is based on such an application.

The P2P-based content repository system enables building the vision of an enterprise-wide wiki as a shared knowledge space and a shared structure of the content organization. It is both *scalable*

DOI: 10.1002/cpe

and shows *good* performance—its major functions are *reconfigurable* to enable a policy-based approach for the content management. However, the most important feature of the system is that it supports *fault-tolerant* and *consistent* content management: as, once content is stored in the system, it shall not be lost. This raises the challenge to coordinate concurrent activity in a dynamic P2P environment and to protect the consistency of created artefacts to keep content up-to-date across geographically distributed locations.

The remainder of this paper is structured as follows. First, the scenario of a P2P-based wiki is described. Next, the background and related work of the approach are given. Subsequently, the system's overall architecture is shown. Thereafter, the major P2P building blocks are introduced. Finally , the approach is evaluated to conclude the paper.

SCENARIO: A P2P-BASED WIKI FOR INTRA-ENTERPRISE KNOWLEDGE MANAGEMENT

As it is common today for enterprises to be present at various globally distributed locations, the need for a shared platform arises to support the collaborative knowledge management. A P2P-based wiki may improve intra-enterprise content management.

Corporations with their organization in many different units show complex structures regarding the number of domains or management of knowledge content. For example, each of the participating departments may maintain its own view of the enterprise world. Usually, a unit represents an organizational-related or product-related task, as accounting or marketing. In different units, however, different content vocabulary may be used.

The usage of a wiki promises to combine the sharing of inter-enterprise knowledge with low administration efforts. From a technical perspective, a *wiki* basically represents a network-based information collection. A wiki's visual representation shall be designed using some template scheme, which defines place holders for the actual content. The actual content shall be selected according to some rules on-demand and it shall be integrated within the relevant part of a corresponding wiki page dynamically at runtime.

Figure 1 illustrates the basic architecture of a centralized wiki system. It is the challenge for a P2P-based system to distribute content management functions and storage but to preserve consistency in the face of concurrent requests.

The scenario assumes that more and more projects require collaboration of geographically distributed persons to exchange content data, or rather knowledge; these persons may belong to different departments, which demands for collaboration across hierarchical boundaries—a drawback of centralized client–server-based systems [2]. P2P content management shall simplify knowledge cooperation by administrating content in one virtual place. This way, it shall facilitate the dissemination of content to all the interested parties. Thereby, the inherent degree of distribution shall be transparent to users.

Technical requirements

In the following, the scenario is analysed regarding requirements at (i) content support, (ii) system function support, and (iii) P2P support.

DOI: 10.1002/cpe

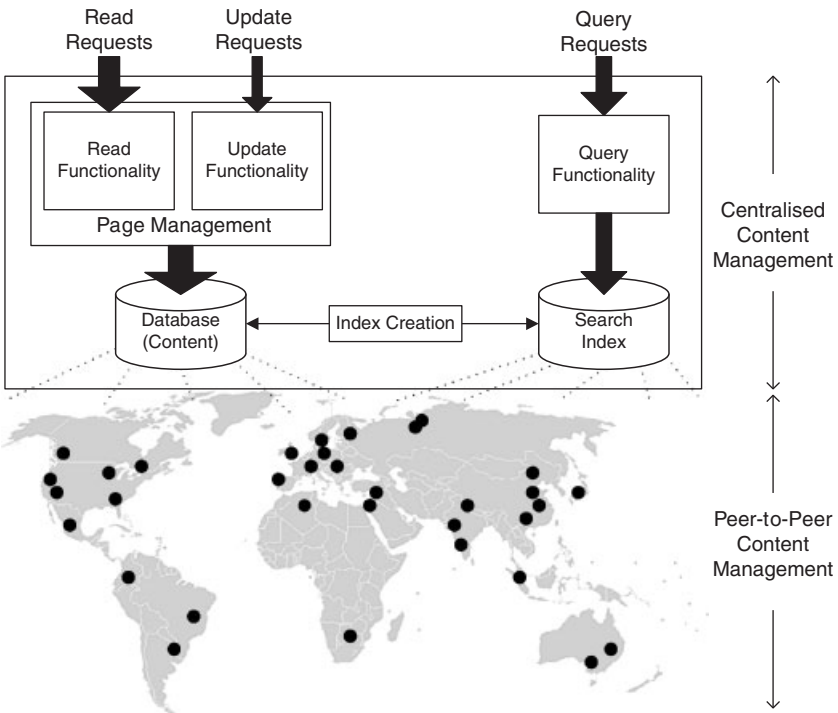


Figure 1. Towards a P2P-based wiki architecture.

At content level, administration of the content shall support the dealing with huge amounts of wiki pages and potentially large media files. For example, a wiki page shall be composed of human-readable, simplified markup syntax. In addition to its textual information, it shall be able to embed files, for example, static content like pictures or streaming multimedia. It shall also enable the usage of *flexible* storage policies for coping with different types of content: for example, different methods may be used to store small-sized, text-based content or large-sized, multimedia content.

Each wiki page shall have a unique identifier and may have cross-references to other pages building a tree-like content structure to allow basic navigation. Hereby, a single page may be divided into several sections to store its current content information and links. A page may be even configured as a kind of symbolic link to redirect all of its read requests to another page. In addition, content may be decorated with meta information such as keywords or *tags* representing authors and other categories. For example, a tag as some freely chosen user-generated metadata refers to a certain aspect of a content object. Multiple tags shall allow some content to belong to more than one category—which is a limitation of the traditional hierarchical organized content.

At the functional level, the life cycle of wiki pages is assumed to be characterized by continuous modifications: new pages may be *created*, existing pages may be *read* or *updated*. The corresponding tags may be dynamically created and may change over time. The scenario assumes that users are required to see an ever-processing view of the shared content, even if high-level conflicts

DOI: 10.1002/cpe

occur: *versioning* shall be enabled to provide mechanisms to detect such conflicts and to support their resolution. Hence, the update of a page shall result in the creation of a new page succeeding the previous version. Once created, a certain version never changes. For each page, some history structure shall exist to link all the versions together to allow the tracking of changes. Change tracking shall support push-based *notifications*, if changes apply to content of interest. The employment of an *access control* mechanism shall allow for user authentication to support the enterprise-wide or department-wide modification of content. While a single version never changes, the editing of shared content may result in concurrent modification requests: a *locking* primitive shall enable the exclusively blocking of content against undesirable update access. Query functionality shall support the passing of tags or keywords to *search* for pages. Thereby, the query part of the architecture shall be separated: it shall use a separate search index generated from the text of pages periodically.

At P2P level, the system shall be *self-organizing* to handle continuous arrivals and departures of peers; for example, as a result of failures. It needs to provide a decentralized method to determine the placement of content, as the physical location of content may change regularly.

Business benefits

From a social point of view, a wiki is formed by a community, which wants to share its knowledge. The scenario claims that it is a great opportunity for an enterprise to employ the wiki concept as a shared intra-enterprise method to exchange and to manage knowledge: the effective management of available knowledge is a deciding competitive factor for enterprises [2]. Access to the relevant knowledge and its utilization are especially desired with respect to shorter product design and development cycles.

However, the scenario identifies the problem to provide access to distributedly stored content and to issue such content in a network of geographically distributed locations with even mobile users. The usage of *collaborative tagging* [9] may help to facilitate and to augment searching for content; it may even increase the possibility of content discovery from the so-called *long tail* [11]. For example, different departments may tag the same content with different keywords—suited for their *own* working domain. It is assumed that *important* content will be usually more often cross-linked with the effect that it is easier to find.

In contrast, a state-of-the-art strategy of intra-enterprise knowledge management may show several drawbacks: the usage of separate knowledge management per department may result in incomplete, inconsistent, and outdated content. Reorganization of department structures may even complicate the conflation of content. From bad to worse, experts who leave the enterprise may leave its often high-value content orphaned: for example, if content is only locally available, it cannot be reused in an efficient way.

The scenario assumes that the amount of the available knowledge content in an enterprise is growing permanently. This growth complicates the management and maintenance of content. A state-of-the-art approach uses a centralized architecture to implement the application logic of a wiki and to administrate its content. For example, the usage of geographically distributed cache servers for content distribution may benefit read requests. However, update requests target the central database. The centralized architecture raises both technical and financial issues for its operator.

DOI: 10.1002/cpe

- From a technical point of view, a central wiki architecture shows modest scaling, because of employed static, central components. This is especially the case in the face of large media data or the great amount of abrupt content requests, so-called flash crowds [12]. Thus, employing a single site would be a bottleneck for the system.
- From an economic point of view, a complete replication of all content at each department site is often neither practical nor cost-effective. However, centralized components would typically constitute the majority of the costs of such system. This raises the question of spreading the infrastructure costs in a fair manner among the departments. In addition, power consumption may impose a restriction as to how large the central location is able to grow in size.

This paper takes the position that the issues described above can be solved by using a P2P-based content repository to implement a wiki application.

BACKGROUND AND RELATED WORK

To our knowledge, no content repository system has been proposed that is flexible enough to implement a P2P-based wiki combining the scalability of the approach with fault-tolerance and consistency properties.

Considering the related work for future Web 3.0, for example, Urdaneta *et al.* [13] indicated a proprietary architecture of a decentralized wiki engine using a gossiping protocol for the data management of dynamic content. However, their proposal is not evaluated and it does not support all the functional properties (e.g. locking, observation) nor the non-functional properties (e.g. flexibility, consistency) of our approach. In addition, there exist several P2P-based systems to enable the collaborative content distribution (e.g. to cache Web pages) [14,15]. But, all these systems focus only on static content and do not consider collaborative working on the dynamic content.

However, the presented system builds on the previous work in the area of content repositories, P2P systems, and group communication, as described in the following.

Content repositories

There exists no uniform definition of a content repository. Bernstein [16] refers to a *repository* as 'a shared database of information about engineered artefacts, such as software, documents, maps, (...) and discrete manufactured components and systems (...). Designing such engineered artefacts requires using software tools. The goal of a repository is to store models and contents of these artefacts to support these tools.' Following Bernstein, a repository is similar to an *object-oriented database* (OODB), as repository systems enable applications 'to store, access, and manipulate objects, rather than records, rows, or entities' [16].

An additional characteristic is that both repository systems and OODB systems have evolved from the trend to drive application functions into the underlying storage system. However, Bernstein identified the differences between the two systems [16]: one major one is the *information model* of a repository system. In database terms, the information model is comparable with a schema for the repository, as it defines a model of the structure and the semantics of the entities that are stored in the repository. The applications that use a repository utilizes its information model to interpret the

DOI: 10.1002/cpe

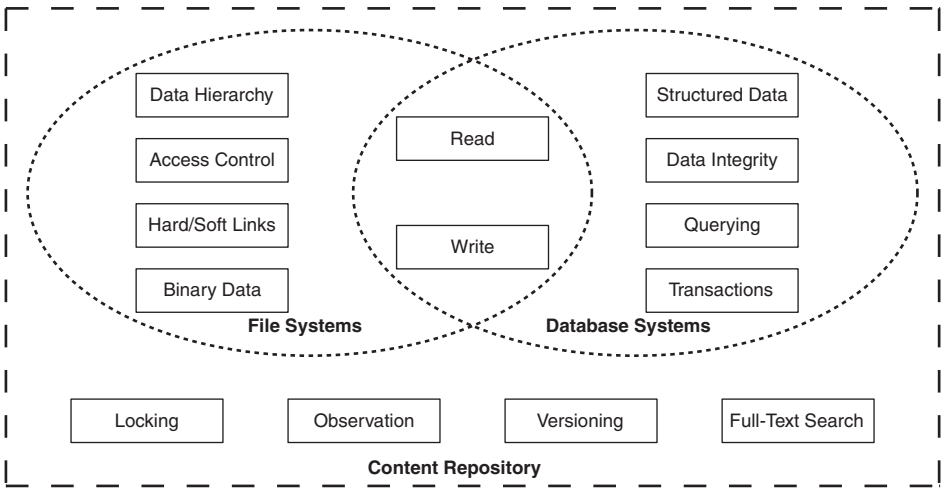


Figure 2. Context of a content repository in relation to file systems and database systems.

repository's contents. This is a difference to database systems, where developers usually assume the information model to be part of the application level. Thus, a repository would support higher-level semantics than OODB systems [16].

A *content* repository can be described as a generic application data store that is able to handle both small- and large-scale data interactions and to deal with structured and unstructured content [17], text and binary data. This way, a repository can be assumed as some high-level information management system that is a superset of the traditional data repositories.

Figure 2 illustrates the potential scope of a content repository. The basic task of a content repository is the providing of content storage. Usually, a content repository combines the basic features of file systems and database systems [18]. For example, file systems typically support hierarchical file storage of binary data and several access control concepts. In contrast, databases enable typically the storage of structured data, provide integrity control, querying functions, and support transactions. Generally, a content repository integrates, in addition to basic storage capabilities, value-added services commonly required by content-centric applications like locking, versioning, or observation.

P2P systems

The P2P model is an alternative to the traditional centralized, client–server computing model: in its purest incarnation, the P2P model treats all of its participants as equal *peers* and has no concept of a dedicated *server* entity. There exist several efforts to define the essential characteristics of P2P systems. Milojicic *et al.* refer to the term P2P as a class of systems and applications that employ *distributed resources* to perform a *critical function* in a *decentralized manner* [19]. These resources may encompass computing power, data, network bandwidth, and presence. The critical function may be distributed computing, data sharing, communication, or platform services. Decentralization may apply to employed algorithms, data, metadata, or all of them. However, requirements may demand to retain centralization in parts of the system or applications.

DOI: 10.1002/cpe

Table I. Scalability and lookup performance of different P2P overlay graphs.

	Reliability	Scalability	Lookup performance
Centralized overlays	*Single point of failure*	$O(\#peers)$	$Constant$
Unstructured overlays	*Redundant lookup paths*	$approx.3–7$	$No\ guarantee$
Structured overlays	*Redundant lookup paths*	$O(log\#peers)$	$O(log\#peers)$
Hybrid overlays	*Redundant index groups*	$O(\#peers/\#partitions)$	$Constant$

A core task of P2P systems is the support for searching of data, more precisely, to assign and to locate data resources among peers. Such mechanism depends on two factors: (i) how the data, and (ii) how the network are organized. The search mechanisms of P2P systems are rather data-oriented, in contrast to the host-oriented ones for traditional networks [20].

P2P search employs P2P overlays, which are logical graphs among the peers. From a logical view, the P2P overlay is situated above the physical network. Data within a P2P system are identified by using indexing methods. There exist different overlays regarding their characteristic topology showing different non-functional properties: for example, Napster's [21] *centralized* overlay, Gnutella's [22] *unstructured* overlay, or Chord's [23] *structured* overlay.

A *structured overlay* topology enforces a *decentralized* indexing structure among the peers to enable a deterministic lookup. All peers share the same namespace, especially, each peer's physical address is mapped to some logical identifier in the namespace using some consistent hash function [24]. Each peer is addressable by such unique identifier and maintains a set of routing information about other peers, for example, its neighbours.

A popular application of such overlay is a *distributed hash table* (DHT). A DHT supports a key-based placement of data objects offering a hash-table interface: in analogy to a hash table's buckets, each peer is responsible for a certain part of the key space; the mapping of data objects to the peers' namespace is done applying the structured overlay's hashing method. However, a drawback of DHTs is their typical focus on immutable data resources when using replication. Another inherent property of the key value-based mapping is the support of a single characteristic per data object. Thus, such mapping does not directly support arbitrary queries as *range queries*. In addition, the keyword-oriented approach fosters hotspot issues for popular areas, and the tight coupling between the overlay structure and the rigid mapping function may cause some overhead concerning the insertion and deletion of data objects, which may be non-trivial under churn [20]. Thus, most DHT-like systems either avoid the difficulty and typically focus on immutable data resources when using replication strategies [25,26], or rather limit the concurrent data resource modifications allowing only one dedicated modifier, the resource's owner [27]. Those rare P2P systems that allow concurrent atomic data operations [28–30] are usually monolithic reinventing the wheel for their storage needs with a focus on their specific application domain, overlay, and strictly on mutable data resources. However, these systems address only part of a content repository's functions, for example, they usually neglect locking or complex querying.

To cope with these issues, the approach of this paper is to use a *hybrid* P2P overlay, which is characterized by combining centralized and structured P2P overlays: (i) the structured overlay represents the basic scope of all peers in the system, that is, the back-end. (ii) The central structure of such overlay is represented by well-defined groups of tightly interconnected peers, that is, P2P service groups. Table I investigates the suitability of different overlays, which motivates the taken approach.

DOI: 10.1002/cpe

One of the major drawbacks of centralized overlays is their reliance on a single index. The *hybrid overlay* overcomes this limitation by distributing routing functionality on groups of indexing peers residing in a structured overlay back-end. Thus, if the lookup schema can be partitioned among several index groups, the routing state for each one shrinks. Regarding the lookup performance, only the corresponding index group needs to be addressed to obtain a data object's location. Then, the data object's host can be contacted using the structured overlay. At its back-end, the structured overlay introduces a consistent mapping between a data object's identifier and the hosting peer. Thus, data load can be distributed across the participating peers. The advantage of such overlay is that each peer is responsible for a certain region in the overlay and that joining or leaving of peers only affects neighbouring peers, immediately. The logical topology of a structured overlay provides some guarantees on the overlay lookup costs achieving high routing efficiency [19].

Group communication

This paper introduces P2P service groups as clusters in the structured overlay back-end. Such groups use group communication for implementing a intra-group message exchange. Group communication has been addressed by many researches for over two decades. The ISIS project initiated the basic work on the group communication paradigm [31,32]. The survey of Défago *et al.* [33] gives an extensive overview of about around 60 known group communication systems.

However, if existing P2P systems like *JXTA* [34] support the syndication of peers to *peer groups* no group communication semantic is provided preventing consistent message exchange, on the one hand. On the other hand, group communication systems are usually not integrated to P2P systems as they do not focus on issues of flexible fault-tolerance in such dynamic environments. In contrast, we present a consensus-based group communication stack for P2P service groups to implement replicated index data structures.

GENERIC SYSTEM ARCHITECTURE

This section introduces (i) the logical view of the content repository system, (ii) its functions, and (iii) its software architecture to typify the substantial assignment of functional responsibilities.

Content repository model

The repository model defines the meta model to identify and structure content data within a repository on a logical level from a user's point of view; it supports to express functional operations on content data. The concrete implementation translates these operations into actual corresponding actions—affecting its used (P2P) storage subsystems. The repository model adopts the *Content Repository API for Java* (JCR) [7,35], which is defined as the open standard to improve the application interoperability [3].

Workspaces and items

The repository model offers a generic, hierarchical content data model, and several levels of functionality for content services on a logical level: a repository consists of an unlimited set of named

DOI: 10.1002/cpe

workspaces; each workspace establishes a single-rooted, virtually hierarchical, *n-ary* tree-based view of content *items*.

Content items are divided into *nodes* and *properties*. Nodes basically provide names and structure to content, which is actually stored in a node's properties. Regarding the content classification, there is no explicit distinction made between real content or meta content. A node may have zero or more *child nodes*, and perhaps zero or more associated properties. Properties themselves cannot have children and are always leaves in the logical tree of aworkspace.

Namespaces and item types

The repository model is generic to support the different types of content items. For their distinction, (i) a namespace concept and (ii) an item-type concept are used. Adopting the concept used in XML [36], *namespaces* may prevent naming collisions between item-type names.

Each node is *typecasted* using namespaced, potentially extensible, names. *Node types* allow the establishment of standardized data-type constraints—for instance, which child nodes and properties a node is allowed or required to have. A node is classified by exactly one *primary node type*. In addition, a node may be equipped with multiple *extra node types*. An extra node type acts as a decorator to add or enforce additional characteristics to those of a primary node type, for example, to mark a node as versionable.

A property must have a certain type to define its expected content *format*. For example, this allows the explicit distinction of *binary* or *string* values.

To support the building of many orthogonal hierarchical views of the same underlying workspace content special property types of weak and strong references are used. Their support shall abstract from a single canonical hierarchy and shall benefit flexible content design strategies.

Variations on item access

Items can be accessed using either *direct* or *traversal* access. In order to uniquely identify each node and ease direct access, it is always referenceable through a UUID, which is unique per workspace. Consequently, a node is independently addressable from its position within the workspace hierarchy. The traversal item accesses targets on walking through the content tree of a workspace, step by step, using (relative) paths.

Content repository functions

In addition to the basic repository model, a content repository is constituted by a set of essential functional building blocks, as illustrated in Figure 3.

Modular decomposition

The modular content repository approach (see Figure 4) considers the horizontal and vertical system decomposition: for instance, horizontally, the distribution degree of content repository functionality regarding the persistent storage support may vary—for example, the storage management for local

DOI: 10.1002/cpe

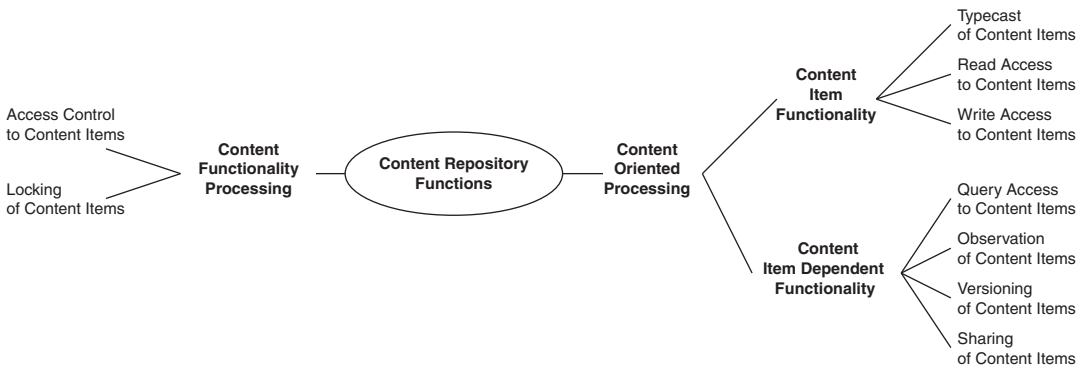


Figure 3. Functional components of a content repository.

or distributed workspaces. Vertically, different modules, for instance, are responsible for different management tasks (as common to horizontal repository functionality).

Each of the architecture's layers is briefly introduced and discussed in the following; whereas, the *persistent storage layer* being a major topic is presented in more detail by the subsequent section reflecting its interaction with transient storage in more detail. Different layers correspond to different major system tasks. This is a novel approach, as usually systems do not distinguish between these layers [3].

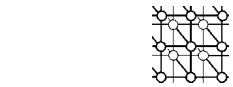
Content application layer: Content applications like a wiki interact through the *content repository* API with the content repository system. That is, the layers below the *content repository layer* may be transparent for it: it does not need to deal with peculiarities of the content storage.

Content repository layer: This layer represents the mapping of the logical repository model to corresponding system modules. For instance, a handle to a *workspace* can be provided via a *session*, received from the *repository* through login to some user credentials—these typically consist of a username and a password to determine the user's access rights. A session represents a long-term connection between a content application and the content repository system; it basically acts as a container to record content item modifications to transient, in-memory storage. In contrast, a workspace represents the persistent storage layer.

At its core, the *repository subsystem* implements several registries and managers, which are further organized in different subsystems:

- The *nodetype registry* is responsible for the storage and retrieval according to typecasting of content items.
- The *namespace registry* deals with the support for the namespace concept.
- The *session* subsystem basically uses a *transient item state manager* to cope with an item's transient state per session. Once a content item is read by a session, it is cached by its transient item state manager. Thus, modified items are only visible to the same session, that is, in its transient storage. In addition, such item state manager is responsible to interpret and resolve a path to an item, or to automatically expand a namespace prefix and to store the full namespace in the repository.

DOI: 10.1002/cpe



- The *workspace* subsystem uses several managers to deal with the repositories' functional building blocks. It depends on the nodetype registry and the namespace registry to create consistent items in persistent storage. It uses (i) a *query manager* to support query access to content items, (ii) a *version manager* to support versioning of content items, (iii) an *observation manager* to support observations of content item changes, and (iv) a *sharing manager* to support sharing of content items. All these managers use the *persistent item state manager* to actually obtain read and write access to a workspace's content items—that is, to get an actual

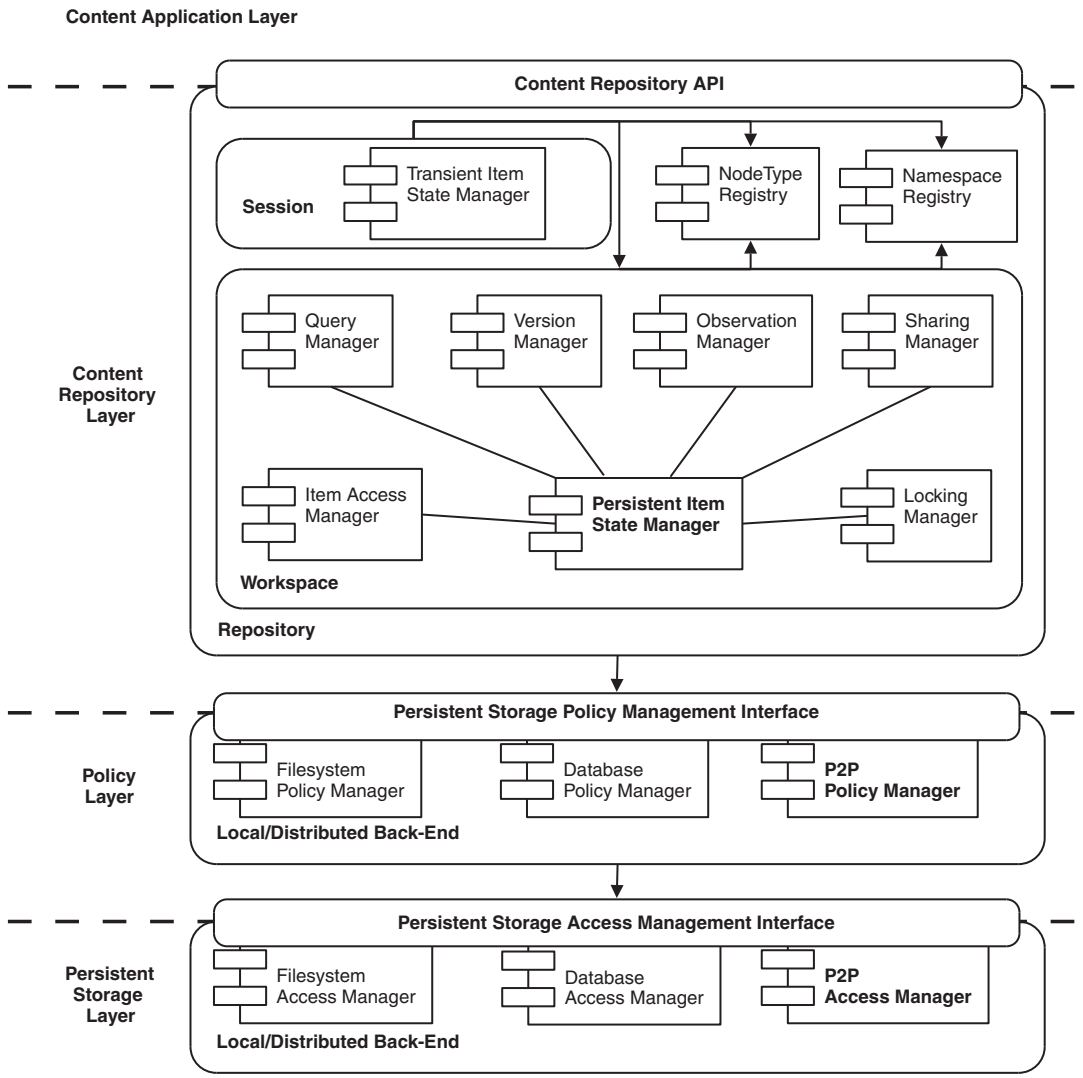


Figure 4. Layered architecture of the modular content repository decomposition.

DOI: 10.1002/cpe

content item view of persisted data; the *persistent item state manager* plays a central role in this subsystem. It represents the connection between the workspace scope and the used persistent storage back-end subsystem; it encapsulates the logic to actually store and retrieve content-item data using path-base or UUID-based addressing—always using corresponding *policy managers* of the *policy layer*. A persistent item state manager is statically configured per workspace; it is able to distinguish between metadata and content data management. A persistent item state manager shall trigger the observation mechanism, if interests in the corresponding item changes exist—usually reflected by some *access manager* of the *persistent storage layer*. This shall enable an observation manager to asynchronously subscribe for changes in a workspace. A *locking manager* and an *item access manager* use a persistent item state manager to enforce their functions, that is, the support of locking and of access control for content items. Accordingly, the persistent item state manager needs to obey such enforced restrictions.

If changes made in a certain session shall be persisted to a workspace, there may be different storage access managers available, for example, a P2P access manager; however, a *policy layer* may be installed above the *persistent storage layer* to enable the additional configuration management.

Policy layer: The policy layer comprises a subsystem to deal with local and distributed persistent storage back-ends; such a subsystem administrates the scope of different storage policies that may be used by the content repository layer to actually access the persistent storage layer. Therefore, it uses *policy managers* matching corresponding *access managers* of the persistent storage layer. There exists a one-to-one relationship between a policy manager and an access manager.

As illustration, the usage of a P2P policy manager enables the definition of potentially fine-granular policies at P2P-data level—rather than on item level. Thus, each type of content or rather content instance may have its own policy; some examples of storage policies in P2P-case may include (i) the life of content, that is, if content shall be stored infinitely or temporarily; (ii) the actual storage location of content, that is, if content shall be stored at a specific peer or if content shall be dynamically moved to another peer if some dedicated peer has not enough storage space left, and (iii), the replication factor of content data resources.

Persistent storage layer: The persistent storage layer defines the subsystem to deal with local or distributed persistent storage at data level. It is indirectly usable by the *persistent item state manager* of the *content repository layer* by exposing a generic *persistent storage access management interface*. Using this interface, several *access managers* for persistent storage may be used, for example, the P2P *access manager*.

Such P2P access manager supports a mapping between a workspace view of content at item level and a raw data view at back-end storage level; thus, it is necessary to use some interpreter to recognize raw data as content items, that is, to retrieve item semantic from raw data resources.

The following section focuses on the system modules, which mainly interact with and are affected by persistent storage management.

Persistent storage management

The modules of the *workspace* subsystem of the *content repository layer* interact with the *persistent storage layer* by the usage of the generic *persistent storage access management interface*—neglecting the *policy layer* for the moment. Together, the subsystems are able to support lookup,

DOI: 10.1002/cpe

Table II. Workspace-supporting operations of the persistent storage access management interface.

$State_{Item}$	`load`	($UUID_{Item}$)
Void	`store`	($State_{Item_1}$, $State_{Item_1}$, .. $State_{Item_n}$)
Boolean	`exists`	($UUID_{Item}$)
Void	`delete`	($UUID_{Item_1}$, $UUID_{Item_2}$, .. $UUID_{Item_n}$)
$State_{Item_1}$, $State_{Item_1}$, .. $State_{Item_n}$	`query`$^{+}$	(*language*, *statement*)
Void	`registerObserver`$^{+}$	($Listener$, $Path_{Item}$, $Type_{Event}$, $Scope$)

search, and modification of the persistent content items using some (distributed) access structures. They represent the content repository's major internal components to deal with the persistent storage of content items.

Regarding the persistent storage management, an important goal is the support of flexible fault-tolerance strategies. Accordingly, suited modules of the *policy layer* may be added on the top of *storage access managers* to support various levels of data replication, for example.

Considering the functional scope of the *persistent storage layer*, Table II states the major operations of the *persistent storage access management interface* regarding the linking of the workspace subsystem. The operations basically reflect *raw data processing* at *system level*. However, the support of two additional operations is defined as optional: (i) `query` and (ii) `registerObserver`. Supporting these two optional operations shall enable to increase the overall system performance by pulling functionality down to tailored methods as offered by the hybrid P2P back-end.

The interface relies on the concept of an item state ($state_{Item}$) to act as container for content items: that is, workspace modules use such states to persist essential information of its functionality as metadata. For example, a *query manager* is able to annotate certain keywords to support full-text search; a *version manager* is able to annotate the version information, or a *locking manager* annotates locking information, for example, the existence of a valid lock.

Considering the *persistent item state manager*, an item state shall reflect the item's workspace name (path) and it's UUID:

- As every item is addressable by a UUID, the `load` operation is responsible to read an item's state from the persistent storage.
- Accordingly, the `store` operation is responsible to persist a set of one or multiple item states. Such item states may reflect the corresponding item-lock or rather item-unlock efforts; in addition, it shall be assumed, that (i) during the processing of an item's state corresponding observation events may be triggered asynchronously; and (ii) item states may be analysed and be indexed for query purposes according to their type.
- The `exists` operation basically verifies the existence of a certain item in the persistent storage.
- The `delete` operation is responsible to remove a certain set of items from the persistent storage.

DOI: 10.1002/cpe

- The optional `query` operation enables a more sophisticated access to persistent storage and provides a generic search interface: it expects the denoting of the used query language and the actual query statement. If successful, the operation shall deliver all item states that match a query.
- The optional `registerObserver` operation supports a workspace's *observation manager* and allows to register a certain listener for a certain *path*—being notified if a certain *event* occurs in shallow or deep workspace *scope*.

Usually, an *access manager* of the *persistent storage layer* conceptually consists of two functional modules: a *metadata manager* and a *data manager*.

Metadata management: A *metadata manager* represents the logical level to deal with the metadata information. It is responsible to administrate all of an item's meta information that is relevant to workspace functionality, such as lookup support, query support, observation support, or locking support. It stores a *path to* UUID and a UUID *to path* two-directional mapping to support the lookup of items, or some kind of *index data structure* [37] to support rich queries. Thus, such metadata reflects the system's item structure, but potentially excludes actual data (or the item contents), which are administrated by a *data manager*, respectively. For example, the P2P back-end system supports full-text search by delegating metadata management to special P2P service groups (indexing groups).

Data management: A *data manager* shall persist content data (blocks) for given addresses: it basically controls I/O *operations* for a given data store. Thus, a *data manager* may be used to implement some raw content data (*blob*) storage. Such manager operates at a very low level and does not need to understand all the complexities of the repository's operations, but essentially just needs to be able to persist and retrieve a given datum based on its identifier. For example, the P2P back-end implements data management on the top of its structured overlay.

Flexible content item policies

The content repository system is intended to support different content storage policies in a flexible manner. For example, such policies may reflect how content may be actually persisted and accessed. It supports different granularity level hierarchies to be built by grouping aggregations of objects to represent larger objects (collections); regarding the granularity level, data objects may be restructured and build from atomic values on demand. Thus, content data must adhere to some global uniform semantics to deal with and to ease content integration—specified by storage policies.

Therefore, an *item type* concept serves to formulate content item policies in a flexible manner. More precisely, a workspace's *policy manager* may interpret a node's *extra types* to select and apply the suited policy per back-end storage. The policy is accordingly annotated to the item's representing data resource. An extra type allows an item to be marked as a kind of special—for example, to mark it as being *versionable* or *more precious*—at content application layer. However, the applied policy may be transparent, as it is applied at *policy layer*. Thus, the node types may be used to annotate contents with type information and to enable their individual storage. Such policy support enables the flexible adjustment of its parameters to implement the different design goals.

DOI: 10.1002/cpe

In the following, some examples of semantics that may be settled by a policy are given:

- A node resource enables to actually embed property resources—thus, their values, too. This facilitates flexible policies, which may actually embed property resources containing *small* values, but place external links for property resources containing *large* values. The policy may state a certain threshold value per corresponding *storage access manager*. This may even allow a *storage access manager* to split up such item unit if the limit is exceeded dynamically at runtime.
- Replication control allows for the determination of the degree of replication and of the placement of replicas. Regarding the level of fault-tolerance per item resource, the said policy may determine the number of replicas per storage back-end. Under certain circumstances it is desirable to expose these details to the *content application layer*; for example, to allow administrators to control such replication scheme.

Internal peer architecture

A peer's internal structure is represented by a generic peer service architecture, as depicted in Figure 5.

A peer's service architecture basically consists of two major components, a *local host abstraction* and a *local service container* [38]. Hence, a peer is made up of hardware and software resources.

Local host abstraction

The *local host abstraction* serves as design element to represent the local system view of a peer. However, a general classification of a peer is difficult, as there exists a wide variety of resources that may be aggregated across peers. For instance, one approach to classify these, is in terms of resources offered by some physical peer device, as CPU processing power, bandwidth constrained network connections, energy consumption, or primary and secondary storage. In addition, each peer usually shows a certain probability to be on line and available to the system. Each peer, however, provides a limited number of local hardware resources; and in contrast to software services, these resources cannot be copied or transferred over a network.

Local service container

The *local service container* follows a service-oriented system design approach to support the dynamic service deployment. Every peer is modelled as a service providing access to the different computational resources of its host. Similarly, each peer provides a container to host other services. This motivates a flexible service model optimized for the dynamic domain of P2P systems. Services may be dynamically integrated into a peer's local service container using a mechanism for dynamic service integration [39]. The services can be divided into three different layers—according to their functional scope:

Local service layer: This layer provides services with some kind of *local* functional scope. A *local storage access service* offers access to a peer's local storage (devices); for example, some key-based storage. A *local network access service* provides messaging on the top of a peer's

DOI: 10.1002/cpe

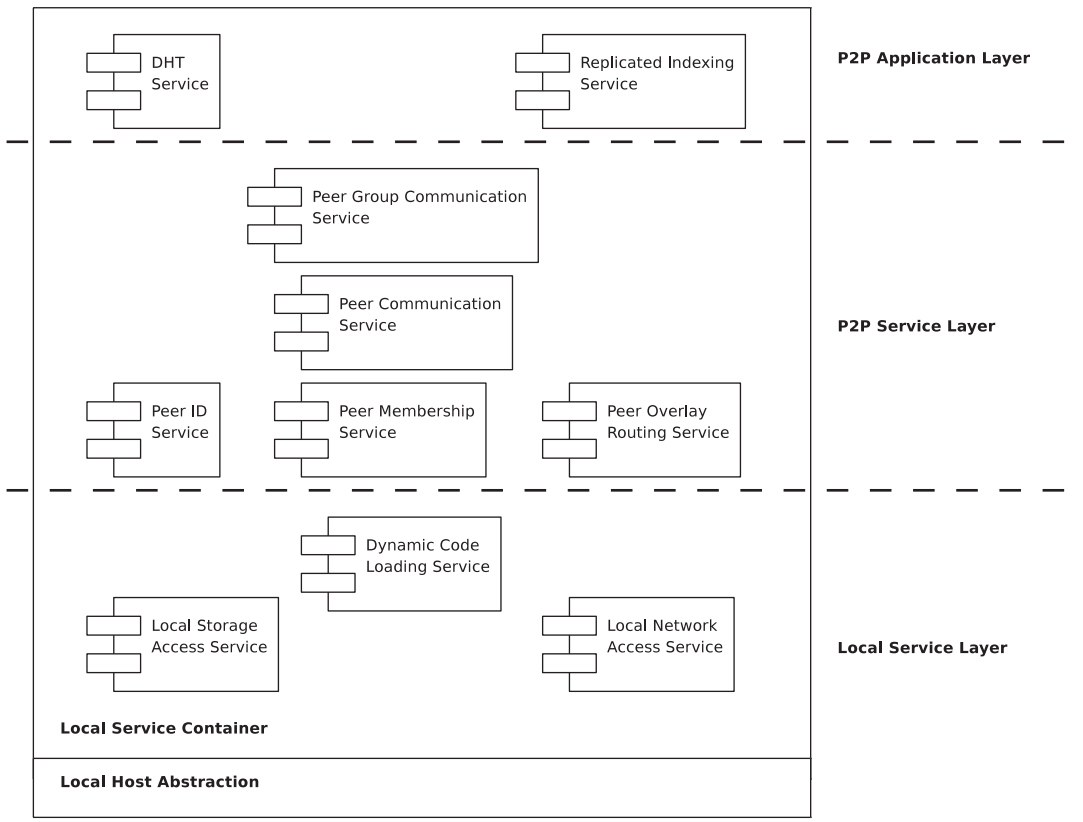


Figure 5. Peer service architecture.

local connections to physical networks; for example, the support of message-communication using TCP/IP or UPD/IP. As mentioned, the *dynamic code loading service* is a facility to integrate additional services to a peer's local container dynamically at runtime.

P2P service layer: This layer provides services with some kind of *distributed* functional scope—the services shall enable to build a P2P-based communication network. In order to interact with each other, a peer needs to offer a set of such essential P2P service functionality and common interfaces. For example, a *peer* ID *service* assigns some unique identifier per peer instance. Such service is required by a *peer membership service*, which manages the joining and leaving of peers, or the transparent updating of its physical network address. Both of these services are used by a *peer overlay routing service*, which implements a certain P2P overlay routing algorithm to create a distributed overlay routing structure using the assigned peer identifiers. The *peer communication service* uses such routing service to enable peer messaging independent from an underlying physical network—with the help of the assigned peer identifiers. A peer group communication service enables peers to syndicate into groups, and send and receive group messages—similar to group communication. The aim of syndicating peers into groups is to enable services that are collectively

DOI: 10.1002/cpe

provided by such groups as a whole (group services), rather than provided by individual peers (peer services). Peers may join or leave a group considering some access policies. However, the internal group management should not be visible to the outside, as well as the internal of services. For instance, the implementation of fault-tolerant group services may enable to tolerate crashes of group members.

P2P application layer: This layer contains the various P2P applications that may be implemented on top of the other layers. The peer architecture enables choices of multiple service implementations for each layer, and P2P applications may be combined with various P2P overlays without any modification. For example, a *replicated indexing service* enables a fault-tolerant storage of an index data structure to implement the metadata management part of a storage access manager for a content repository—tailored for hybrid P2P overlays.

P2P BUILDING BLOCKS

This section presents the major building blocks of a P2P access manager. It explains (i) the hybrid overlay structure, (ii) P2P service groups, and (iii) the used method to achieve flexible atomic data management for the P2P back-end.

Hybrid system structure

Figure 6 shows the layered architecture of the hybrid approach offering a two-tier hierarchy: (i) generally, each computer node is represented by a peer in the *DHT layer* of the system, the latter being the *structured* aspect of the system's overlay. (ii) The *P2P service group layer* enables different peers to syndicate into groups, which form the *central* aspects of the system's overlay.

This basic concept of a hybrid overlay is the basis to use P2P service groups as building blocks to implement a persistent content storage back-end using the decoupling of metadata management and data management; as already mentioned, P2P service groups are introduced as concepts to support implementing a replicated index to administrate the metadata of a content repository's workspace.

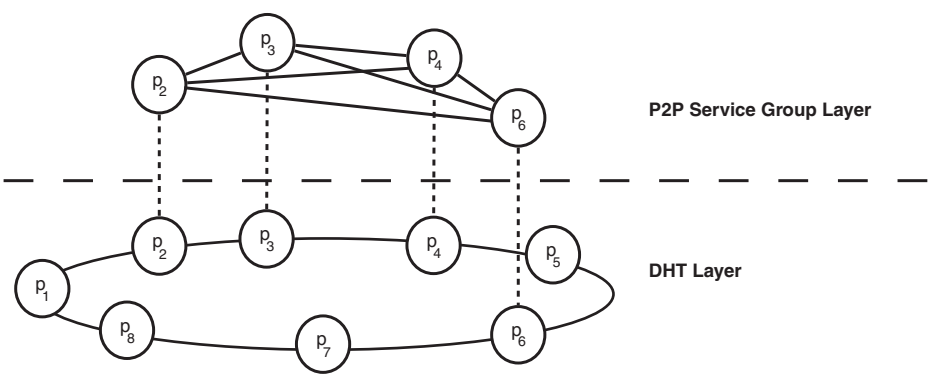


Figure 6. Hybrid overlay architecture.

DOI: 10.1002/cpe

Regarding this implementation, there exist two different roles of system peers: *indexing peers* and *storage peers*. As already indicated, the reason lies in the nature of DHTs: DHTs provide a simplified put–get interface to efficiently store and retrieve content resources by keywords, for instance, UUIDs; unfortunately, their support for more sophisticated queries, such as range queries and semantic queries over large data sets, for example, a workspace's distributed item tree is difficult concerning certain non-functional requirements [38].

Storage peers enable the key value-based storage of a workspace's content resources. Basically, every system peer not acting as an indexing peer is considered a *storage peer*. Therefore, each storage peer offers local storage capacity to store the actual content data of the repository; that is, the data management part is *delegated* to the DHT layer. Each peer hosts a service container with a set of standard services to manage the service execution and to integrate services dynamically at runtime; this mechanism enables equipping regular storage peers with indexing service capabilities to act as indexing peers or to remove these capabilities again.

The major metadata of each workspace is, however, concentrated by corresponding indexing peers, which are implemented by a P2P service group. Indexing peers may provide an *advanced* querying interface for sophisticated queries, as required by the support of the *persistent storage access management interface*. To enhance their *internal* communication latencies, indexing peers use the mentioned group communication module to maintain a separate pool of connections to other indexing peers—in addition to *normal* DHT connections. In the following, the set of indexing peers is referred to as *indexing group*. A workspace's metadata may be injected at an arbitrary peer of the indexing group. Afterwards, it is internally disseminated through a (group) communication protocol. Thus, an indexing group acts as a kind of an *island* within the DHT layer to support certain operations more efficiently: a workspace index may be distributed or shared among those peers and involves all system peers—a policy may be used to determine the size of such indexing group. Using such policy, an indexing group is able to adapt its size and to integrate new peers to support the resilience and load sharing. However, the integration of new peers may require the assistance of consistent migration decisions of a workspace index; that is, which information should be transferred to a new member.

Reconfigurable P2P service groups

P2P service groups provide a manner to break the symmetry of peers and to exploit their *diversity*. Intuitively, a *peer group* represents some kind of central component in the P2P overlay by concentrating a *certain* service to a *certain* set of selected peers. Hence, it basically represents a group of peers dedicated to execute a common *group service*. A P2P service group may be constructed *ad hoc*, as soon as a group service is ready to be deployed in the system. Thereby, such P2P service group is *reconfigurable*: (i) peer group memberships can change dynamically at runtime; in addition, the offered service can be (ii) deployed and (iii) reconfigured dynamically at runtime applying some *policy*. The life cycle management of these groups includes the discovery of *suitable* peers. Hence, a P2P service group represents some kind of partitioning scheme of the world of peers; for example, to foster the performance, communication, or logical locality. In addition, the cooperation of peers may provide reliability of the service execution. However, peers of a service group may take certain roles identifying their responsibility regarding the group formation and the execution.

DOI: 10.1002/cpe

As hybrid-overlay aspect, P2P service groups are designed to run on the top of a structured P2P overlay. P2P service groups serve as the method to implement a distributed, replicated, and fault-tolerant repository index. An important aspect of the concept is the establishment of a consistent inner-group communication mechanism. Therefore, such service group uses a generic consensus module as an inner-group communication component [40] to support the building of replicated state machines. The special aim of replicated P2P state machines is to benefit repository functions working at *deep* operational scope of a workspace's distributed content tree: the replication of relevant content item metadata on different peers is a useful redundancy for improving the availability. But such multi-peer replication has the potential to foster the performance, too; on the one hand, the selecting of a *nearby* group peer to serve a query request may result in shorter service time. On the other hand, fewer peers and communication messages may be involved in such query process within a group; for example, no overlay lookup costs may be required to send messages between replicas.

The challenges for such P2P group communication system comprise: (i) the *consistent* adapting of a group to dynamically changing members, (ii) the support of service-specific ordering semantics on the order of delivery of messages, and (iii) the providing of several fault-tolerance semantics applying some policy. The usage of distributed consensus algorithms is an established way to implement a common group communication system, which supports total message ordering. However, the system additionally provides mechanisms supporting its (re)configurability. Unlike other systems, a P2P service group communication instance can be configured to work with different failure models and low-level communication protocols without changing the service part. In addition, different failure models, protocols, and their runtime parameters (for example, time-out limits) can even be reconfigured dynamically at runtime without losing consistency, especially in case of failures. Reconfiguration at runtime promises for a service to adapt to access patterns and environment conditions for gaining the optimal performance and fault-tolerance at the same time. As the reconfiguration method is transparent to the service logic, it may be even initiated automatically by the underlying system.

For the implementation of an infrastructure for fault-tolerance, this has two important impacts: first, best service quality will only be obtained if the infrastructure is flexible to allow *service- and environment-specific tailoring*—depending on the requirements of a certain service and the properties of the environment. Second, the infrastructure has to support *flexible runtime adaptation*, as both the needs of the service and the properties of the environment may change dynamically at runtime. Faced with the need of an adequate support for tailoring and runtime adaptation at the P2P group-communication level, existing systems for group communication could not meet the requirements regarding these issues. Active replication requires totally ordered multicast semantics within various models of fault (for example, crash-stop, crash-recovery, or Byzantine), which are optimized for the specific service requirements and environment properties. The proposed P2P group communication system uses an encapsulated consensus module to obtain total order. Many specializations of this generic module exist and thus provide an ideal basis for application-specific tailoring. These specializations include the seminal Paxos algorithm [41] and existing variants for low latency as well as for fail-stop, crash-recovery, and malicious failure models. Group members may transparently decide to replace the instantiation of the consensus module with another one to tolerate different kind of faults or to adjust parameters that influence the performance. In addition, the low-level communication mechanisms may also be dynamically configured, for example, applying TCP, SOAP, or TLS.

DOI: 10.1002/cpe

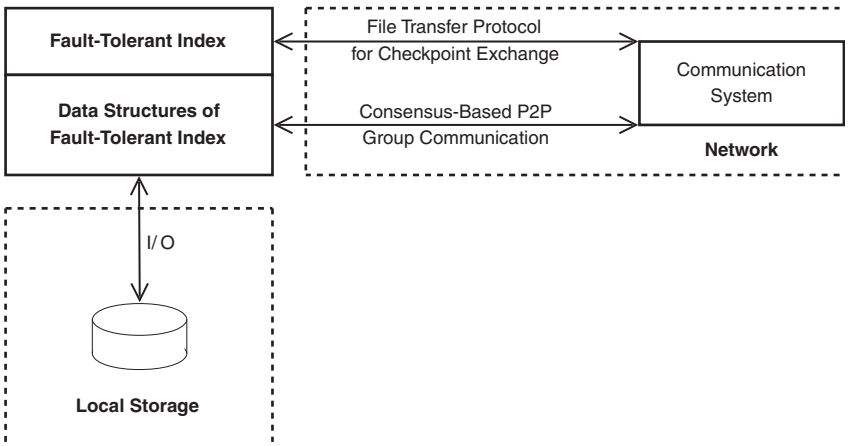


Figure 7. Two dimensions of a fault-tolerant workspace index.

Figure 7 shows the two *dimensions* of a fault-tolerant workspace index: (i) a local dimension and (ii) a network dimension.

Locally, each indexing peer maintains a view of the fault-tolerant index itself and the data structures to actually create it. For the latter, an indexing peer is intended to administrate different types of metadata; for example, the item namespaces, the mapping from paths to UUIDs, or relevant inverted indices. All such data structures are kept in an indexing peer's transient local memory and persistent local storage. The usage of such replicated index enables a reliable update of an indexing peer's state without the risking of inconsistencies in case of peer failures: therefore, the generic consensus module is used—accessed by the P2P group communication system.

An indexing peer's local data structures of the fault-tolerant index shall reflect a historical record of critical metadata changes. Modifications of the data structures need to be made persistently, however, before being exposed to external peer requests. In order to increase the availability and fault-tolerance, such workspace index is replicated among multiple indexing peers and a client's request is served only after flushing the corresponding record to disk, both locally and remotely—using the consensus protocol instance. In order to increase system throughput, several operations may be batched together. An indexing peer is able to restore its state by replaying the relevant data structures. In order to keep their history *small*, however, a checkpointing mechanism may be used, if the size reaches a certain limit. Thus, by restoring the latest checkpoint from local disk may only require a limited number of index records. Outdated peers may access up-to-date information by using some file transfer protocol for checkpoint exchange.

To sum up things, the described architecture is quite flexible as different overlay protocols may be used at the first tier as well as at the second tier. Especially at the intra-group level, P2P service groups may use different ways to establish the group communication. For example, if the size of a group is quite small (less than 20 members), each member could track all other group peers and may use group communication mechanisms to implement the intra-group communication. If the group was larger (about 100 members), selected group members may be used to track all other group members. Finally, if a group is large (about 1000 members), structured overlays may be used

DOI: 10.1002/cpe

to implement the tracking and intra-group communication. In the context of this paper, however, only quite small groups are assumed.

Interface

Considering the persistent storage management, the relevant workspace-supporting operations of the *persistent storage access management interface* are supported by the system.

The `Group` component is the principal module of an instance of the P2P group communication system: it shows the interface that is visible to a (peer) service. The interface offers, for example, operations (i) to *join*, (ii) to *leave*, and (iii) to reconfigure a P2P service group by adjusting group policies. Concerning *policy changes*, all reconfiguration actions are subject to the group's consensus and are delivered to all group members in total order. To support communication between group members, the component offers methods to send and to receive (group) messages.

Modular group communication structure

The modular design of the internal architecture of the *consensus-based reconfigurable group communication system* is outlined in Figure 8 [40]. The `Group` component represents the core of any P2P service group. It implements the external interface that is visible to a service application and

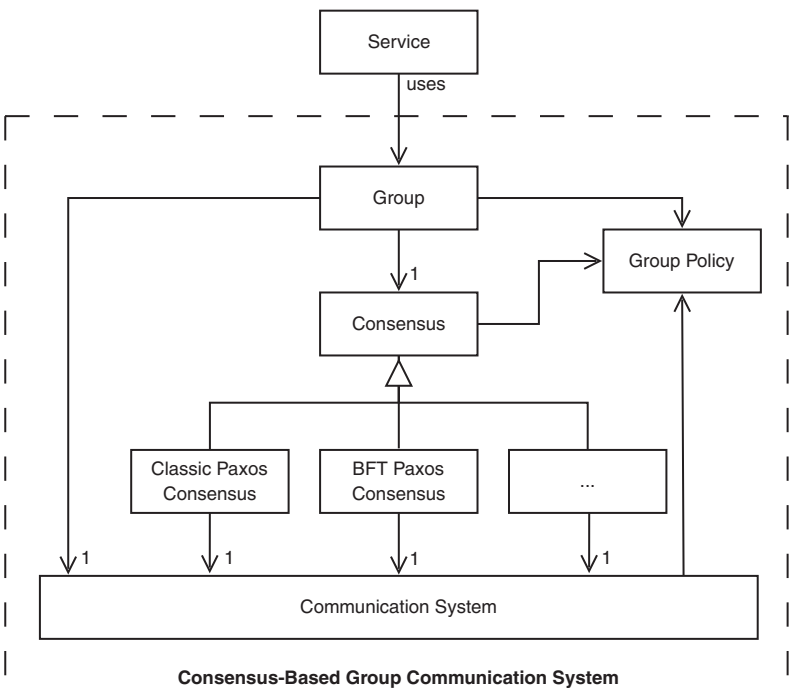


Figure 8. Modular structure of the reconfigurable consensus-based group communication system.

DOI: 10.1002/cpe

internally uses the `Consensus` component to obtain the total order of all group messages between its members. The generic design of the `Consensus` component supports a variety of implementations, each with different *quality-of-service* properties. Both the `Group` and the `Consensus` component use an instance of the `Communication System`, which provides *low-level* messaging between participating peers. The configuration of all the three main components is described by a given *Group Policy*. This policy, internally represented as a list of key-value pairs, is defined at group creation time and may be changed at runtime by a dynamic reconfiguration process. For example, a policy may define which peers are allowed to interact with a certain P2P service group.

The `Communication System` component encapsulates the specific low-level mechanisms that are used for communication: it provides network independent addressing between group members using *peer IDs* without requiring an actual P2P lookup, handles message queueing, and re-establishes connections after failures. This component represents a communication abstraction and fully supports reconfigurability; depending on the available network abilities, different variants like plain TCP/IP or UDP/IP connections, tunnelling via SOAP/HTML, encrypted TLS channels, or the use of existing hardware multicast mechanisms can be supported [40]. The `Communication System` offers an asynchronous (non-blocking) sending primitive to the using components: each message is tagged with a message type to allow a direct delivery to the appropriate entity. The group policy defines the instantiation to be used as well as corresponding parameters, like time-outs for connection re-establishment.

Flexible atomic data management for the P2P back-end

As already mentioned, the hybrid overlay uses a DHT back-end. Combined with additional replication strategies such systems promise high availability for published data resources. A common approach to enhance fault-tolerance in P2P systems is to store a certain data resource instance replicated at different physically located peers, called its *replication group*. However, regarding the support for *atomic data operations* replication comes at the cost of maintaining data consistency: an atomic data operation on a certain resource has to be consistently applied to all of its replicas. In this case, the existence of several replicas is crucial and raises the challenge if a data resource could be modified concurrently.

The system uses *DhtFlex* [42] to enable flexible atomic data operations for replicated data; that is, a distributed algorithm that is trimmed for such a highly concurrent and fluctuating environment, where peers may fail with high rate, so-called *churn* [43]. DhtFlex bridges the gap between the mentioned requirements of a DHT service and the benefits offered by a structured key-based routing overlay. DhtFlex acts as a generic building block in a modular environment, as illustrated in Figure 9 [42], which is responsible for the complete data management including replication handling.

The query model of DhtFlex supports simple *read* and *write* operations for data items that are uniquely identified by some *key* (UUID). It uses techniques that extend a DHT in order to deal with the requirements emerging of supporting the content repository functionality. Hereby, DhtFlex supports both immutable as well as mutable data resources and offers flexible consistency strategies for atomic data operations. For example, once a certain version of a wiki page is defined, it remains forever unchanged within the corresponding version chain.

DhtFlex imposes an annotates data resource concept to typify replicated data. This allows the differentiation between data items and the efficient dealing with both immutable, as well as mutable

DOI: 10.1002/cpe

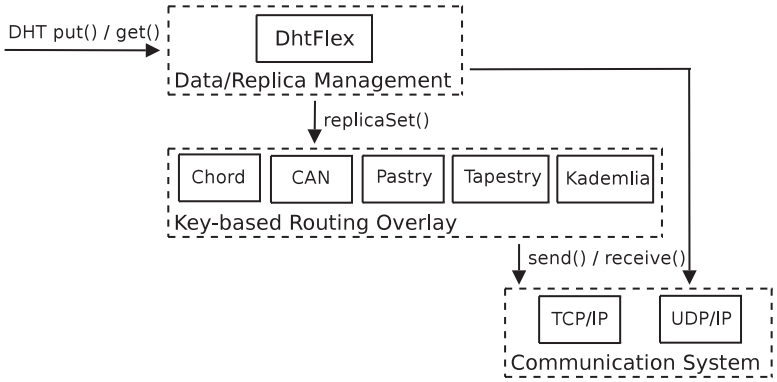


Figure 9. Interactions of the major building blocks of DhtFlex's system environment.

data resources. Especially for the latter, DhtFlex is able to provide strong consistency guarantees enabling atomic DHT *put* and *get* operations. Therefore, it exploits techniques of Leslie Lamport's famous Paxos algorithm to coordinate the *recast* process of a data resource's replication group. DhtFlex serializes concurrent put and get requests over the master of a replication group in order to accelerate these operations. It allows system growths to large scales and updates to be made from anywhere in the system.

Interface

The DhtFlex algorithm associates each data item with a unique key or id; it offers the two common DHT operations: *put*(*id*, *value*) and *get*(*id*). The *put*(*id*, *value*) tries to commit a value for a certain id to the system: DhtFlex locates the peer that is responsible to host the id and publishes the value, if successful. Considering the underlying structured P2P overlay, a hash function is applied on the id to generate a fixed size identifier; this identifier is used to determine the peers that should be responsible for serving the id. The *get*(*id*) operation tries to determine the id's responsible peer in the P2P overlay and returns the corresponding current valid value, if successful.

Partitioning strategy

DhtFlex employs the structured overlay part for the data resource placement. The content item view of the DhtFlex approach is to regard these as uniquely addressable, single uniform objects. It is easy to see that this model naturally maps to the used key-based routing overlays.

Thereby, DhtFlex offers a DHT abstraction from the routing of messages between peers: for example, a peer that sends a message usually does not know the destination peer *a priori*; a key is used to identify the target peer rather than an explicit destination address. This is a great difference in comparison with the traditional routing mechanisms, for instance, as used in IP routing. The abstraction layer is responsible to forward a message *msg* that carries *key* towards the corresponding *root* peer of this key in the P2P overlay. For the common overlay protocols, the root is that peer, which possesses the numerically closest matching identifier in comparison with the key.

DOI: 10.1002/cpe

However, as a dynamic peer environment is assumed, network conditions can change over time. As a result, a key's corresponding root may vary. Peers that enter or leave the network demand the used overlay protocol to adjust responsibilities for affected key ranges; for instance, gaps in the overlay resulting from down peers need to be closed. As DhtFlex does support crash-recovery, as well as crash-stop failure models, it is able to exploit positive dynamics of a structured P2P overlay, where peers may take over the key range of a failed one. The worst-case scenario of such maintenance operations occurs if the overlay cannot be repaired resulting in the overlay breakup. For example, network partitioning may lead to such islanding problem, where an overlay splits into independent sub-overlays not interlinked with each other. DhtFlex may be able to detect such failure situations in order to thus support the consistency of affected data operations.

Replication strategy

As shown in the previous section, the exposed DHT abstraction basically maps keys to values; thereby, a value may be an arbitrary object or an item represented as data resource, which may be replicated and persistently stored. An object is retrieved by using the key under which it was published. DhtFlex uses replication in order to ensure the high availability and durability of administrated data resources. Thereby, it supports a flexible degree of replication that allows an adjustment per data resource type.

However, if a peer leaves the system, for example, by crashing, its administrated data resources become unavailable. A replication mechanism increases the data availability by storing data at several peers. But, in the face of concurrent modifications mutual consistency of replicated data resources may be violated, some replicas may not be up-to-date. The requirements of content repository functionality demands for DhtFlex to be able to get the current valid replica.

A replicated data item is independent of the peer on which it resides and may be regarded as virtual. This applied *virtualization* enables DhtFlex to employ structured overlay routing as partitioning strategy. Thereby, DhtFlex manages all replication functions; the overlay is accessed only to conduct the necessary information to construct a *replication group*. A replication group configuration is a set of peers that are responsible to administrate a certain replicated data resource. The size of such a set is defined by the resource's replication degree. A replication group of size n consists of one master and $n-1$ replicas.

Regarding the replication model, DhtFlex implements a *primary-copy* replication pattern [44] per replication group: a replication group's master is used to serialize and apply all updates to a mutable data object.

In order to benefit the partitioning strategy, DhtFlex uses the unique key of a data resource to configure the corresponding root in the overlay as master. Accordingly, DhtFlex targets to fill the replication group set with the *available* $n-1$ peers succeeding a root in the overlay, the $n-1$ *root successors*. Hence, a replication group of size n shall contain those $n-1$ peers that are relevant to become a root for the key after network conditions change. Regarding fault-tolerance aspects, these $n-1$ peers are ideal candidates to place the replicas of a given data object.

The master of a replication group is responsible to ensure the replication factor for the data resources that fall within its key range. That is, in addition to their conservation in local storage, the master needs to replicate the resources to the remaining replicas. This implies, that changes on resources have to be propagated to all replicas in order to ensure the consistency.

DOI: 10.1002/cpe

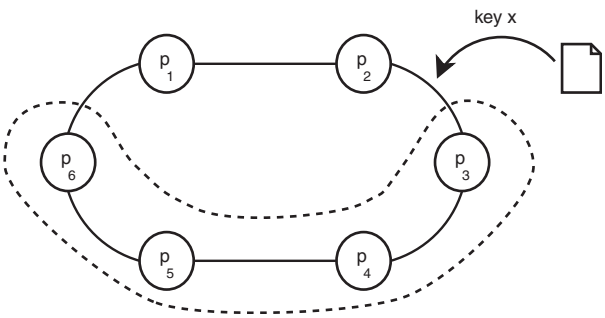


Figure 10. Combination of replication strategy and partitioning strategy.

The replication strategy in combination with the used partitioning strategy is exemplified in Figure 10. It shows a replication group consisting of one master peer p_3 and three additional replica peers: the master p_3 replicates the data object for key x at peer p_4, p_5, and p_6. Hence, p_6 stores values that fall into the ranges $(p_2, p_3]$, $(p_3, p_4]$, $(p_4, p_5]$, $(p_5, p_6]$. As explained, the employed structured P2P overlay allows each peer to determine, which peers should be contained in the replication group for a certain key.

FLEXIBLE CONTENT REPOSITORY FUNCTIONS

This section illustrates how the introduced P2P building blocks are used to implement the flexible content data functions. Therefore, (i) an appropriate content mapping is indicated, and (ii) it is shown how the functionality of the *content repository layer* can be implemented using the P2P system at *policy layer* and *persistent storage layer*.

Content mapping

As explained, the introduced mapping between content items and data resources is flexible to benefit the separation between *metadata management* and *data management*. The mapping enables to use P2P service groups to implement a *metadata manager* for persistent workspace storage, and the DHT layer to implement a *data manager*. In addition, the approach supports fine-grained data resource replication for both layers.

As already indicated, the P2P service group method enables to implement an indexing group to administrate the metadata structures (index). Accordingly, a corresponding manager of the *policy layer* is able to use the (re)configurability features to enforce policy requirements for an indexing group—for instance, to determine the amount of replicas for a replicated workspace index, or the size of a property value's replication group in the structured overlay back-end. To ensure robust execution of the system functions in the case of peer failures, replication is used to allocate *identical* data resources or data structures at different peers. Policy information can be used by an access manager at the *persistent storage level* to process such resources accordingly.

DOI: 10.1002/cpe

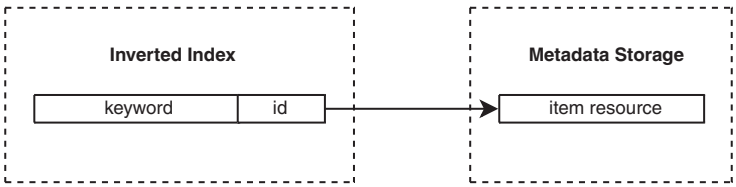


Figure 11. Data structures of an indexing peer.

The system uses an *item bundle concept* to keep the content mapping manageable—that is, to define which data resources may be bundled together to be effectively administrated by the hybrid system. Considering *node resources* and *property resources*, the approach uses the following scheme:

- The node resources and the property resources—representing the metadata information of a workspace—are administrated as a kind of *local* bundle unit by each replica of the workspace's corresponding indexing group.
- The actual property value, however, is usually stored as *remote* data value administrated by the structured overlay back-end. Therefore, such value is referenced via a *remote storage location* in a property resource. That is, the location links to the affected peer(s) in the DHT layer.

A peer of the DHT layer needs to provide a local key value-based persistent storage as data structure to support the data management. These structures represent the mapping of *remote storage location* to actual data value.

A replica of an indexing group uses several additional local data structures—based on reverse indexes—to benefit mechanisms for persistent *metadata management*, as depicted in Figure 11.

For example, each replica locally indexes a *node resource* by its *id* entry—as a node's UUID is sufficient to guarantee a unique addressing in the workspace context; as each property's *name* is unique per node, the combination of the parent node's *id* and the property's *name* allows a non-ambiguous indexing of each property resource.

Thus, the logical tree structure of a workspace can be locally represented according to the administrated content items in *metadata storage*, which is replicated among the participating indexing peers. To benefit a query-based lookup, an inverted index may be used to serve as a *short-cut* between indexing information and item resources (see Figure 11). As illustration, an inverted index can be used to match a certain keyword to a set of relevant items. For example, such data structure benefits implementation of full-text searching for certain property values. These data structures need to be, however, kept consistently to reflect the current logical tree structure. For example, as child nodes or properties of a node changes, all affected entries of the inverted index would need to be updated.

Persistent content storage

The imposed functional requirements on the system interface essentially require to deal with the storage of item resources to support operations such as *store*, *load*, *exist*, *delete*, *query*, and *register some listener*. The P2P system supports these operations for indexing groups using the

DOI: 10.1002/cpe

message-based interface of the P2P service group layer. Additionally, the DHT layer provides a basic put–get interface for key-value pairs.

Regarding the basic architecture of an indexing peer, at the bottom of its stack each replica maintains a local copy of the replicated index data structures. On the top, the next layer represents the fault-tolerant replicated index. To establish such replicated index, replicas communicate with each other using the concrete consensus protocol instance of the P2P group communication system. A consensus instance is used to ensure consistency at the resource level; that is, the modifications at data resource level are propagated as proposal values to establish a total ordering of such operations—thus, these are exchanged between the members of such P2P service groups in a consistent manner. The protocol instance ensures that each replica's local data structures consist of identical sequences of entries. The possibility of using an indexing peer's local data structures facilitates the task to implement the metadata management considering persistent storage operations.

If these operations need to be atomic, the consensus-based group communication system of an indexing group is able to support this by submitting such operations as a single value. DhtFlex is used to ensure consistency for the DHT layer. Thereby, the approach supports a consistency model similar to the relaxed so-called *close-to-open consistency model* [45,46]. The major benefits that such an approach provides is that temporarily made changes on local items need not be committed to the network until the modifying operation is done and write access is closed. This implies, that once an item has been locally accessed or opened, a peer need not remotely check with the network if that item has been modified in the meantime by another peer. It is consistent to locally cache an item as long as it is opened and until it is closed.

The indexing group approach can be used to support the close-to-open model by retrieving the latest item resource via a *retrieval* operation once the item should be locally opened; then, such item resource is kept as a cached copy by the *content repository layer* until access is closed. All succeeding requests to an item's potential properties or child nodes can be satisfied using information from the cached copy.

Considering the reading or *loading* of an item, Figure 12 illustrates such retrieval process using the hybrid architecture. (1) First, an access manager uses its local peer instance to pass a query or path statement for one or multiple items to the corresponding indexing group—that is, to one member.

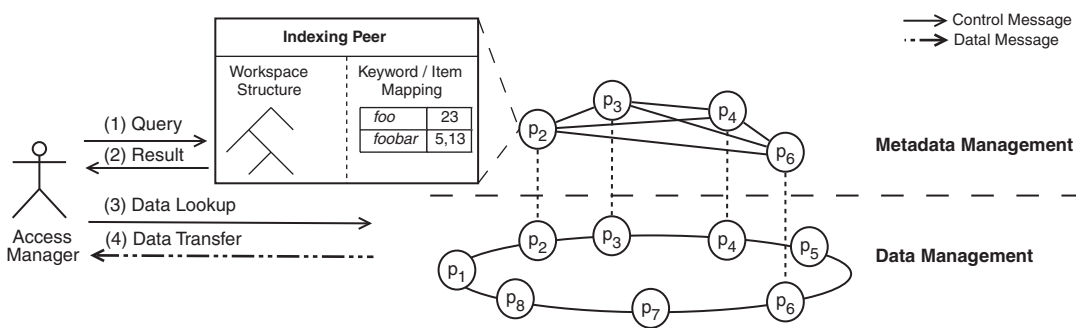


Figure 12. Content data retrieval in case of a hybrid overlay.

DOI: 10.1002/cpe

If valid, this member promotes the request as consensus proposal to the group's communication system. Thus, the group is able to operate as a control instance regulating access policy; for example, to control which peer is allowed to pass a request. (2) If the indexing peers eventually decide on the query statement, the contacted member processes it against its local workspace structure and its local keyword—item mapping—always respecting the total ordering of consensus decisions. The result of the processed query—that is, the matching item resources—is returned to the requesting peer. (3) In case of property resources being returned, they may contain links to data which is actually stored by some peer in the DHT layer. Thus, the remote storage location may be contacted. (4) The actual data transfer is handled by the requesting peer and the corresponding storage peer of the DHT layer. It is worth mentioning, that only the last step involves transmitting of a *larger* data message. The previous steps require only the exchange of *smaller* control messages. Thus, the actual data transfer is decoupled.

If an item should be modified, a peer's locally cached copy is updated—at content repository level—to reflect the changes; hence, write efforts and corresponding changes are locally buffered by a *session* before being stored to the network in order to minimize local write latencies. Finally, once item access is closed, all cached changes are flushed to the hybrid network and tried to be committed. Considering the support of write or *store* operations, valid type restrictions need to be respected. Usually, all actions that may modify an item's state are expected to *load* the according item resource, first. Then, the item can be constructed and thus type consistency checks are enabled at *content repository level*—at item state level. Generally, a *writer* peer is assumed not to fail during its writing process to complete the corresponding actions.

Figure 13 shows the inner process of an access manager if an item should be stored. (1) First, the item resources are constructed and passed to a member of the workspace's indexing group. *Large* property values are not transferred but kept at the local storage. It is the task of

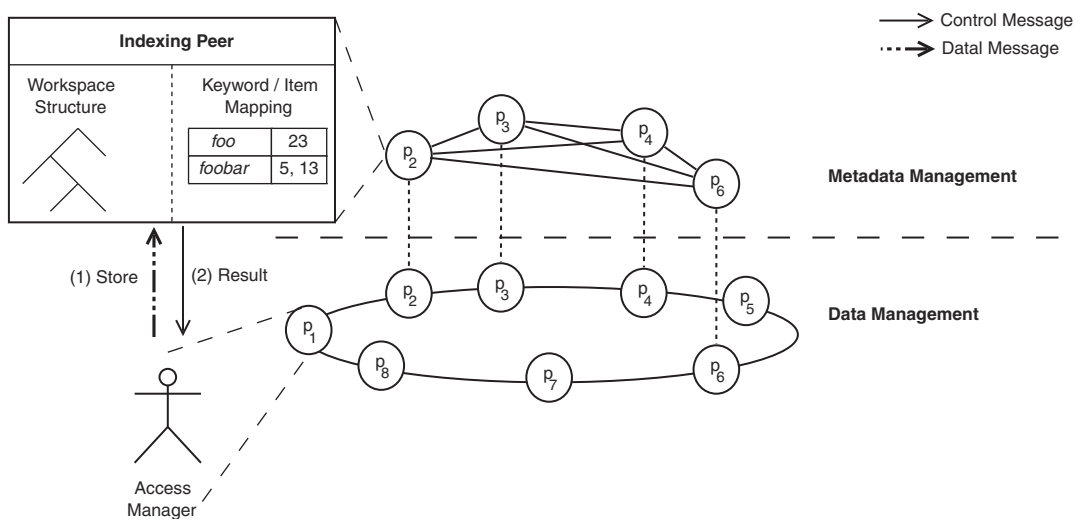


Figure 13. Content data storing in case of a hybrid overlay.

DOI: 10.1002/cpe

the access manager to specify a suitable remote storage location concerning the DHT layer and apply policies, for example, demanding the replication of property values at the DHT layer. (2) The indexing group tries to process the storage request. If successful, an acknowledgement is returned.

As an additional step, that is, if *large* property values are involved, for example, an access manager may either store such values using a specified *remote storage location*, or it may use DhtFlex—utilizing the structured back-end as some kind of *decoupled* address space for target-oriented lookup of data values; that is, the retrieval of such values is based on UUIDs (metadata) rather than on concrete physical addresses.

Regarding the support of content repository functions, an indexing peer can use its local data structures, however, to process shallow as well as deep operations. For example, the support of a *query* operation and a *locking* operation requires to basically rely on a replicas local workspace structure and local item mapping—always respecting the established total ordering among operation requests. Accordingly, *observations* can be implemented by performing matching tests reacting on the adding, removing, and modifying of affected item resources. This supports basic event notification mechanisms that allow the triggering of a notification, if a suited node resource for a certain path in the virtual tree of a workspace is stored. The subscriber of an observation event may be known by every replica; however, only the contacted indexing peer may actually inform the subscriber to prevent the unnecessary network traffic. Of course, if that replica fails, some kind of a handover mechanism is needed. The support of versioning is kind of straightforward, using an indexing group and the explained *load* and *store* primitives.

EVALUATION

This section evaluates the presented system considering (i) its overall architecture, (ii) reliability, (iii) scalability, and (iv) performance properties.

Overall architecture

The architecture is evaluated considering the introduced wiki scenario.

The system enables (i) mapping wiki pages to its item concept, and (ii) different bundling of a page's data resources (as illustrated in Figure 14): for example, (small-sized) textual content may be attached to a wiki page's representing *node resource* as *property resource*, on the one hand. On the other hand, different pages may share common (large-sized) multimedia contents, and different transport protocols may be used to retrieve them *on demand*.

The *repository model* supports UUID-based addressing of wiki pages; basic navigation is supported by the *workspace tree*. *References* support cross-linking to other pages and symbolic linking for redirecting read requests. Tags may be modelled by *extra node types* to allow the multiple classification of wiki pages.

The system supports flexible content repository functions in a P2P environment: for example, DhtFlex enables to represent a wiki page as *mutable* data resource but to keep a single version of it as *immutable* resource. Resources may be *locked* to prevent the undesirable update access.

DOI: 10.1002/cpe

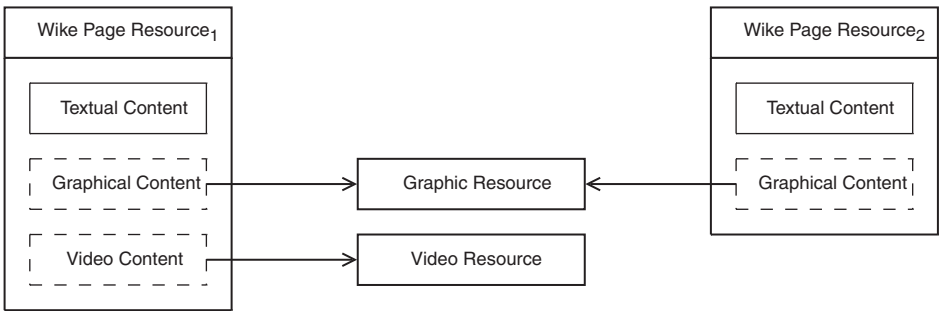


Figure 14. Mapping a wiki page to the item bundle concept.

In addition, indexing peers may enable change tracking by supporting (deep) push-based *notifications*. As each indexing peer keeps a replica of its corresponding workspace's metadata, querying for content is supported too.

At the P2P level, the presented methods support *self-organizing* of peers at persistent storage layer, for example, executing consistent movement of data resources as a result of failures. In addition, the system enables *dynamic* integration of (heterogeneous) peers.

Reliability

The structured P2P back-end ensures reliability using replication as redundancy scheme: that is, a certain number of identical copies are stored at different peers. For example, a certain version of a wiki page is stored as immutable data resource. Thereby, a resource's replication factor ρ influences its availability—thus, the value of ρ should be set appropriately depending on the demanded degree of availability.

The analysis assumes the *worst case*, that is, no reconfiguration actions occur intermediately. In addition, it is assumed that a peer's availability is independent—that is, peers fail independently—and that all peers show an identical (average) availability α_{peer}.

DhtFlex requires a single copy of the ρ replicated (immutable) data resources to be available for progress successfully. Thus, the probability P_{fail}—the failure of a data resource's whole replication group—is given by the following equation:

$$P_{fail} = P(\textit{all } \rho \textit{ replica peers fail}) = P(\textit{one replica peer fails})^{\rho} = (1-\alpha_{peer})^{\rho} \quad (1)$$

A resource's replication factor ρ can be adjusted depending on the desired availability aim, as stated by the following formula: $\rho = log(P_{fail})/log(1-\alpha_{peer})$. Figure 15 depicts the probability P_{fail} to actually lose an immutable data resource—depending on different values for α_{peer} and ρ: one observation is that comparatively small values for the size of a replication group suffice to reduce the probability of losing a certain data resource significantly—usually, reaching a *certain* limit an additional increase of ρ does not significantly reduce P_{fail}. Considering the scenario, for example, if $\alpha_{peer} = 0.7$ is assumed, *good* availability may be achieved by using four replicas ($P_{fail} < 0.01$).

DOI: 10.1002/cpe

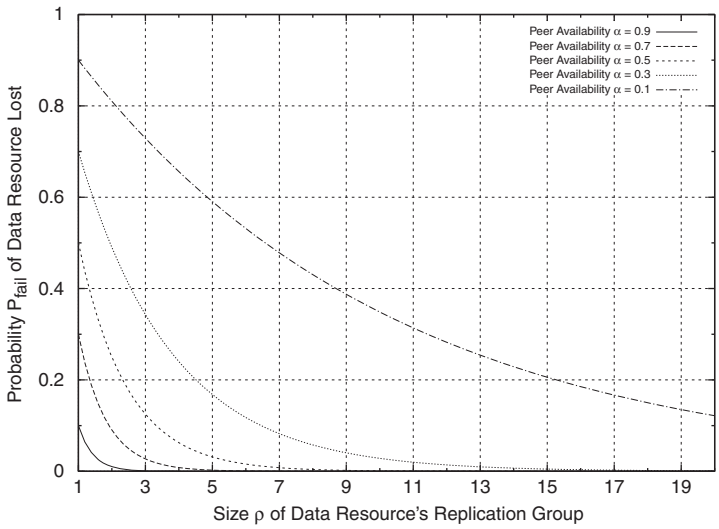


Figure 15. Worst-case probability an immutable data resource is lost.

Scalability

This section discusses scalability for the P2P back-end and the P2P service groups. Indexing groups represent critical parts in the hybrid overlay: that is, as these play a major role for supporting rich queries, the system selects only peers showing *good* properties as group members; for example, considering such peer's hardware resources high processor throughput, and large primary and secondary storage space is demanded. In addition, group members should communicate at the high network connection speed to reduce message latencies (for example, by being located in physically close distance).

P2P back-end

The *total* data load of a P2P system is defined as the sum of the data loads of all participating peers. The *data load* of each peer refers to the amount of data resources a peer is responsible to store locally. This section evaluates the distribution of content data using practical evaluation. It investigates the distribution of 1000 data resources with varying replication factor ρ to a simulated network of 1000 peers—using Chord as overlay protocol.

According to the wiki scenario, the data resources represent the *1000 most viewed articles* of the English version of Wikipedia [10] in August 2008 to indicate if DhtFlex's partitioning strategy *is suited*:

- SHA-1 is used as hash function to create both peer identifiers and data resource identifiers.
- Each peer is responsible for a certain (key) segment of the overlay: thereby, although the hash functions are used to achieve *good distribution* of peers in the overlay, the size of segments may vary.

DOI: 10.1002/cpe

- The following schema is applied to create the unique name for an item resource representing a certain Wikipedia page: $\underbrace{en.wikipedia.org/wiki/}_{\text{namespace}}.name_{article}$

Figure 16 shows the results of the experiments: in all cases, the theoretically expected value μ is achieved as a kind of centre of distribution: (i) $\mu=1$ in case of $\rho=1$ (1000 data resources), (ii) $\mu=5$ in case of $\rho=5$ (5000 data resources), and (iii) $\mu=10$ in case of $\rho=10$ (10 000 data resources). In general, the data load on each peer scales *well* with the number data resources and varying values of ρ: no *hotspots* are detected.

On the one hand, increasing the value of ρ seems to level data distribution; on the other hand, the number of peers not storing a resource may be decreased. DhtFlex may ensure load balancing by uniform distribution of peer identifiers and data resource identifiers, where the number of data resources stored at each peer is roughly balanced. To deal with *large* data files, special *storage peers* may be used and only metadata may be committed to the system.

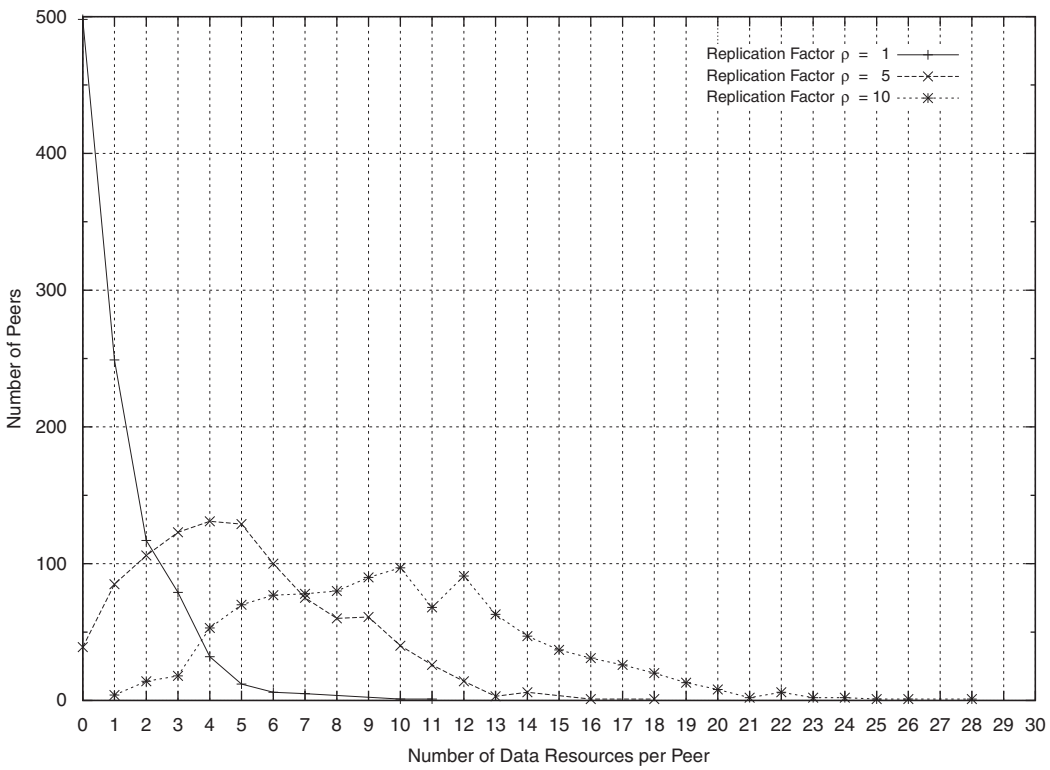


Figure 16. Data distribution of 1000 wiki data resources on 1000 peers.

DOI: 10.1002/cpe

P2P service groups

The scalability for P2P service groups is indicated by a qualitative discussion of the applied data querying strategy: indexing peers support querying for data resources by their *materialized view* of workspace items: that is, each indexing peer may use its *local view* of the workspace tree to process query requests.

- On the one hand, this may increase the lookup performance in comparison to using the structured overlay back-end, exclusively—especially, if the addressed search space (for example, a workspace subtree) is *large*: for instance, the structured overlay back-end is only used to retrieve the actual content data (the query results); that is, data transfer may be decoupled from the metadata management.
- One the other hand, indexing peers are affected by more *request load* than *usual* peers (in the structured overlay back-end) and need to ensure the consistency of their local workspace view. For the latter case, the following section analyses the (traffic) costs for maintaining the consistency.

Performance

This section discusses the performance properties for the P2P back-end and the P2P service groups. Performance analysis for P2P service groups considers (i) local operations of a peer and (ii) distributed operations between multiple peers.

P2P back-end

The performance evaluation of the P2P back-end uses simulation to determine the latency of its operations, that is, to read (*get*) and to write (*put*) the contents of a wiki page. Considering experimental setup, a *King data set* [47] is used to weight communication links between simulated peers. This approach enables to gain estimations of communication costs based on the real measurement data of thousands of Internet hosts[‡].

Immutable data resources: Figure 17 depicts the latency of DhtFlex for operations on *immutable* data resources, for example, representing different versions of a page. The latency of *put* operations is greater than that of *get* operations. Both operations are strongly affected by the costs to perform the P2P overlay routing, which increase with the number of peers—the operations introduce, however, rather constant overhead by themselves. As the *recast* operation does not require overlay routing, its latency is comparatively small.

Mutable data resources: Figure 18 gives the latency of DhtFlex for operations on *mutable* data resources, for example, representing the current valid pages. Again, the latency of *put* operations is greater than that of *get* operations, and both operations are strongly influenced by the costs to perform the P2P overlay routing. However, both *put* and *get* operations add rather constant overhead

[‡]The simulation results neglect, however, messages of different sizes. All shown latencies represent the average value of ten measurements per operation using random keys for *put* and *get* operations. Each data resource is allocated to a replication group of six (different) peers.

DOI: 10.1002/cpe

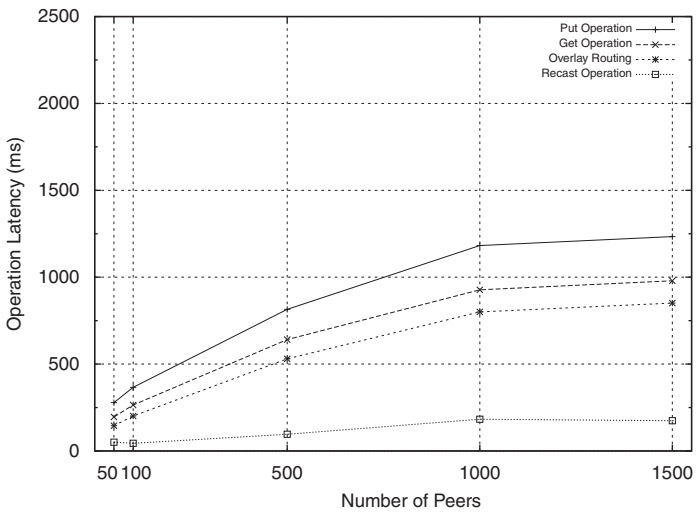


Figure 17. Latency of DhtFlex for immutable data resources.

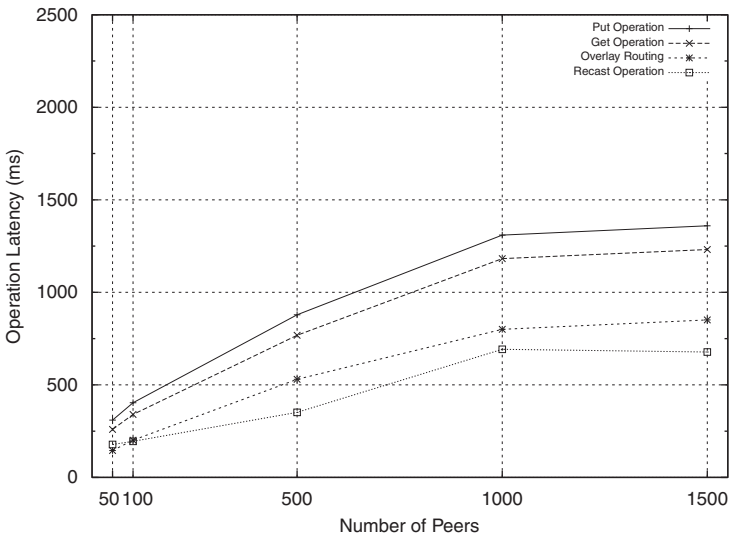


Figure 18. Latency of DhtFlex for mutable data resources.

by themselves. On the one hand, both operations require higher latencies in comparison to operations on *immutable* data resources. On the other hand, the latencies for *put* and *get* operations do not differ significantly in both cases. In contrast, *recast* operations (that is, to consistently reconfigure a replication group) are more expensive for *mutable* data resources.

DOI: 10.1002/cpe

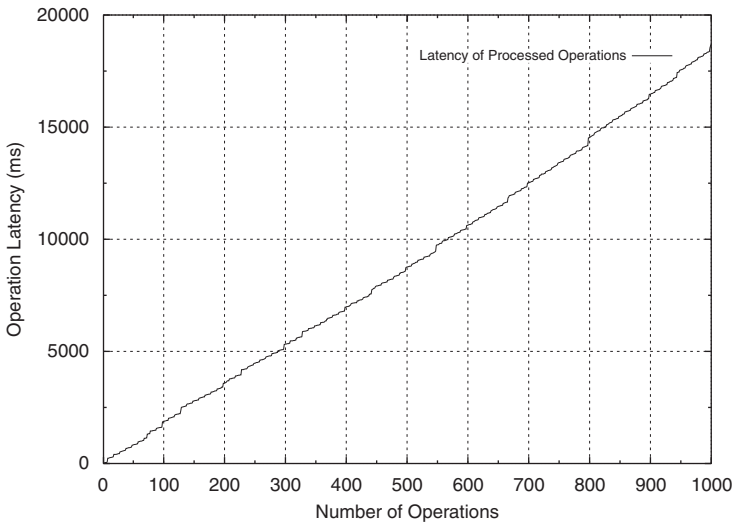


Figure 19. Latency of 1000 local indexing-storage operations.

Hence, DhtFlex is optimized for get, put and recast operations on mutable data resources, in that sequence. These operations are even more efficiently supported for immutable data resources, increasing the overall performance in an employed system.

P2P service groups: *local case*

Practical evaluation applying direct experiments is used to indicate an indexing peer's local performance: that is, to state the latency (i) to index (store) and (ii) to query workspace items. The evaluation considers the processing of multiple item–property bundles representing wiki pages with 3141 *mean bytes per article*[§,¶].

The results of the experiments are given in the following:

- Figure 19 depicts the aggregated latency to locally store and index 1000 item bundles: it is obvious, that the overall latency grows linearly with the number of processed item bundles.
- Using the inserted items, the average *query latency* (considering 100 executed queries) to search one item is in the range of -2 ms.

[§] http://stats.wikimedia.org/, retrieved on 2008/12/28.

[¶] The experiments are executed on an AMD Athlon XP 3000+ (2.09 GHz) machine with 1 GB main memory running Windows XP Professional. The local access manager at the persistent storage layer uses a combination of Java-based Lucene and Apache Derby for indexing and searching content items.

DOI: 10.1002/cpe

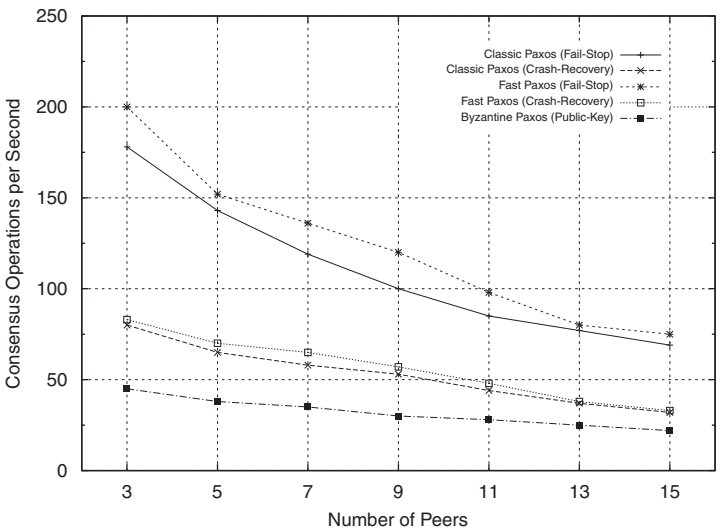


Figure 20. Performance of different consensus algorithms.

P2P service groups: *distributed case*

Practical evaluation applying direct experiments is used to indicate the performance of *intra* group communication as a crucial part of the hybrid approach (without considering the protocol for the structured overlay back-end).

The processing of *normal* group messages is the dominant operation in an indexing group, while reconfiguration actions will typically occur far less frequently. Therefore, the measurements focus on the normal-case efficiency of the reconfigurable component applying different configurations.

The current implementation uses different variants of the Paxos algorithm for the consensus module, which support different failure models and different parametrizations (to optimize latency and message overhead). The analysis examines throughput and latency characteristics of different configurations to illustrate their feasibility[‖].

Performance using different consensus modules: Regarding the performance, the most important factor of the consensus-based group communication system is the efficiency of consensus decisions. Figure 20 [40] depicts the number of consensus decisions per second that the system achieves in relation to the number of core group peers—for different consensus modules: the crash-recovery variants use synchronous writes to the local hard-disk drive as stable storage; for all variants, the parallelism of consensus decision was limited to *five* parallel instances. TCP channels were used for low-level communication between peers.

[‖]In the following, all described direct experiments were executed on up to 15 Intel Pentium 4 (3.0 GHz) machines running Linux (kernel 2.4), connected via a switched 100 Base-T network; the system is implemented in Java (J2SE 1.4).

DOI: 10.1002/cpe

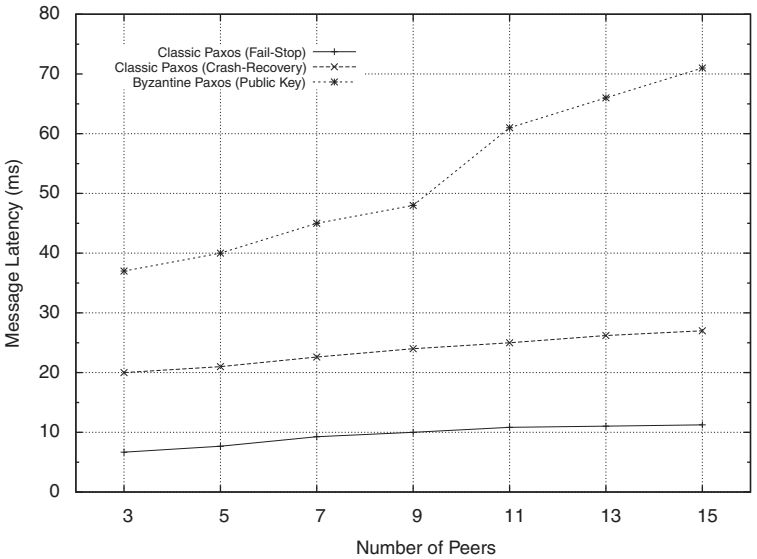


Figure 21. Overall P2P group message latency with different consensus algorithms.

For all depicted algorithms, the system scales quite well with an increasing number of group members. The limiting factor in the stable storage-based variants are the synchronous write operations—all writes have to be flushed to disk immediately before a peer may proceed; that is, as the peers are *close* in the network proximity delays for disk flush may influence the (even dominate) overall latency.

Message latency using different consensus modules: From a persistent storage manager's point of view, an essential efficiency criteria is message latency: for example, the caused overhead to ensure the consistency by synchronization efforts or to include a new group member.

Figure 21 [40] shows this latency for three different consensus variants—depending on an indexing group's size. In the process, all times for the depicted message latencies are averaged over 100 messages sent to group to acquire a consensus decision; the latencies are measured at persistent storage layer, that is, directly using the `Group` component's interface. With each variant, the message latency increases with a growing number of participating group peers.

CONCLUSIONS AND OUTLOOK

This paper presented and evaluated the method to use a flexible P2P-based content repository to implement a decentralized wiki reducing a wiki's creation and maintenance costs: presentation of content shall be decoupled from its background organization and storage location to support the construction of web page incarnations on demand. The decoupling of design management, data structures, and content shall support reuse of content.

DOI: 10.1002/cpe

The system operates different functions to remove central components avoiding single points of failure. For example, it uses content replication strategies to be less vulnerable to the failure of individual network nodes and network connections. The P2P system uses a hybrid overlay and a flexible architecture to implement the required functions in the distributed environment. It is self-organizing to reduce the administration and *ad hoc* integration efforts; it offers the potential to reduce operating central resources like dedicated server hardware. In addition, the approach promises to help distributing content and to reduce hosting costs at the same time: for example, by employing and aggregating already available commodity computing resources, it offers the potential to spread infrastructure costs in a fair manner.

The system ensures the reliability using replication as redundancy scheme—the introduced storage redundancy shows linear growth. For example, in case of immutable data resources, only a single available replica is sufficient to progress successfully. However, crash-recovery and recasting are supported to additionally increase reliability. Regarding the P2P back-end, data load distribution scales in logarithmic order with the number of peers. For P2P service groups, it is emphasized that only *good* peers shall be selected to act as indexing peers.

Evaluation of the performance properties showed that the P2P back-end is optimized for get, put, and recast operations on mutable data resources. However, these operations are even more efficiently supported for immutable resources, increasing the overall performance in an employed system. As a result, the communication costs of the back-end are comparable with that of non-atomic DHTs, in most of the cases. Regarding the performance of P2P service groups: in the local case, a service group peer's overall operation latency grows linearly with the number of processed items. In the distributed case, the performance properties scale quite well with an increasing number of group members.

Regarding the future work, it would be interesting to integrate semantic technologies at the application level, for example, Semantic Web technologies. As a result, decentralized semantic wikis would enable to establish a partnership between human and automated collaborators—a vision of the future industry scenarios. For example, by easing the transfer of informal textual knowledge to more formal structures both people and machines may use wiki content. This may benefit advanced querying over a wiki's distributed content set facilitating interlinking, (re)use, and extension.

REFERENCES

1. GartnerConsulting. The emergence of distributed content management and peer-to-peer content networks. *Technical Report*, Gartner Group, January 2001.
2. Schmücker J, Müller W. Praxiserfahrungen bei der Einführung dezentraler Wissensmanagement-Lösungen. *Wirtschaftsinformatik* 2003; **3**:307–311.
3. Bartlang U, Stäber F, Müller JP. Introducing a JSR-170 standard-compliant peer-to-peer content repository to support business collaboration. *Expanding the Knowledge Economy*: *Issues*, *Applications and Case Studies*, *Information and Communication Technologies and the Knowledge Economy*, vol. 4, Cunnigham P, Cunnigham M (eds.). IIM, IOS Press: Amsterdam, The Netherlands, 2007; 814–821. *Proceedings of eChallenges e-2007 Conference*.
4. Dustdar S, Gall H, Hauswirth M. *Software-Architekturen für Verteilte Systeme*: *Prinzipien*, *Bausteine und Standardarchitekturen für moderne Software*. Springer: Berlin, 2003.
5. True picture of P2P filesharing. *Technical Report*, CacheLogic Research, 2004.
6. Ghemawat S, Gobioff H, Leung ST. The Google file system. *SIGOPS Operating Systems Review* 2003; **37**(5):29–43.
7. Day Management AG. Content repository API for Java^{TM} technology specification May 2005. *Java Specification Request 170*, version 1.0, 2005.
8. Leuf B, Cunningham W. *The Wiki Way*: *Quick Collaboration on the Web*. Addison-Wesley Professional: Reading, MA, 2001.

DOI: 10.1002/cpe

9. O'Reilly T. What is Web 2.0: Design patterns and business models for the next generation of software. *Technical Report*, September 2005.
10. Wikipedia, the free encyclopedia. Available at: http://www.wikipedia.org/ [15 January 2009].
11. Anderson C. The long tail. *Wired Magazine* 2004; **10**:170–177.
12. Stading T, Maniatis P, Baker M. Peer-to-peer caching schemes to address flash crowds. *Peer-to-Peer Systems* 2002; **2429**/**2002**:203–213.
13. Urdaneta G, Pierre G, van Steen M. A decentralized wiki engine for collaborative wikipedia hosting. *Proceedings of the 3rd International Conference on Web Information Systems and Technologies*, Barcelona, Spain, 2007.
14. Freedman MJ, Freudenthal E, Mazières D. Democratizing content publication with coral. *NSDI'04*: *Proceedings of the 1st Conference on Symposium on Networked Systems Design and Implementation*. USENIX Association: Berkeley, CA, U.S.A., 2004; 18.
15. Wang L, Park KS, Pang R, Pai V, Peterson L. Reliability and security in the codeen content distribution network. *ATEC '04*: *Proceedings of the Annual Conference on USENIX Annual Technical Conference*. USENIX Association: Berkeley, CA, U.S.A., 2004; 14.
16. Bernstein PA. Repositories and object oriented databases. *SIGMOD Record* 1998; **27**(1):88–96.
17. Fielding RT. JSR170 overview: Standardizing the content repository interface. *Technical Report*, Day Management AG, 2005.
18. Jackrabbit ZC. die Referenzimplementierung des Java Content Repository. *Linux-Magazin Sonderheft* 2008; (3):18–21.
19. Milojicic DS, Kalogeraki V, Lukose R, Nagaraja K, Pruyne J, Richard B, Rollins S, Xu Z. Peer-to-peer computing. *Technical Report*, Hewlett-Packard Company, March 2002.
20. Kshemkalyani AD, Singhal M. *Distributed Computing*: *Principles*, *Algorithms*, *and Systems* (1st edn). Cambridge University Press: Cambridge, 2008.
21. Napster FS. Available at: http://www.napster.com [30 June 1999].
22. Clip2. The annotated Gnutella protocol specification v0.4 (document revision 1.6), 2001.
23. Stoica I, Morris R, Karger D, Kaashoek MF, Balakrishnan H. Chord: A scalable peer-to-peer lookup service for internet applications. *SIGCOMM '01*: *Proceedings of the 2001 Conference on Applications*, *Technologies*, *Architectures*, *and Protocols for Computer Communications*. ACM: New York, NY, U.S.A., 2001; 149–160.
24. Karger D, Lehman E, Leighton T, Panigrahy R, Levine M, Lewin D. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. *STOC '97*: *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing*. ACM: New York, NY, U.S.A., 1997; 654–663.
25. Rowstron A, Druschel P. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. *SOSP '01*: *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*. ACM: New York, NY, U.S.A., 2001; 188–201.
26. Dabek F, Kaashoek MF, Karger D, Morris R, Stoica I. Wide-area cooperative storage with CFS. *SOSP '01*: *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*. ACM Press: New York, NY, U.S.A., 2001; 202–215.
27. Rusitschka S, Southall A. The resource management framework: A system for managing metadata in decentralized networks using peer-to-peer technology. *Agents and Peer-to-Peer Computing* (*Lecture Notes in Computer Science*, vol. 2530/2003). Springer: Berlin/Heidelberg, 2003; 144–149.
28. Bindel D, Chen Y, Eaton P, Geels D, Gummadi R, Rhea S, Weatherspoon H, Weimer W, Wells C, Zhao B, Kubiatowicz J. Oceanstore: An extremely wide-area storage system. *Technical Report UCB/CSD-00-1102*, EECS Department, University of California, Berkeley, 2000.
29. DeCandia G, Hastorun D, Jampani M, Kakulapati G, Lakshman A, Pilchin A, Sivasubramanian S, Vosshall P, Vogels W. Dynamo: Amazon's highly available key-value store. *SOSP '07*: *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*. ACM: New York, NY, U.S.A., 2007; 205–220.
30. Kubiatowicz J, Bindel D, Chen Y, Czerwinski S, Eaton P, Geels D, Gummadi R, Rhea S, Weatherspoon H, Wells C, Zhao B. Oceanstore: An architecture for global-scale persistent storage. *ASPLOS-IX*: *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM: New York, NY, U.S.A., 2000; 190–201.
31. Birman KP, Joseph TA. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems* 1987; **5**(1):47–76.
32. Schiper A, Birman K, Stephenson P. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems* 1991; **9**(3):272–314.
33. Défago X, Schiper A, Urbán P. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Computing Surveys* 2004; **36**(4):372–421.
34. Microsystems S. JXTA v2.3.x: Java programmer's guide. *Technical Report*, 2005.
35. Day Management AG. Content repository API for Java^{TM} technology specification July 2007. *Java Specification Request 283*, version 2.0, 2007.
36. Bray T, Hollander D, Layman A, Tobin R. *Namespaces in XML 1.0* (2nd edn), W3C Recommendation, August 2006.

DOI: 10.1002/cpe

37. Zobel J, Moffat A, Ramamohanarao K. Inverted files versus signature files for text indexing. *ACM Transactions on Database Systems* 1998; **23**(4):453–490.
38. Gerdes C, Bartlang U, Müller JP. Decentralised and reliable service infrastructure to enable corporate cloud computing. *Collaboration and the Knowledge Economy*: *Issues*, *Applications and Case Studies*, *Information and Communication Technologies and the Knowledge Economy*, vol. 5, Cunnigham P, Cunnigham M (eds.). IIM, IOS Press: Amsterdam, The Netherlands, 2008; 683–690. *Proceedings of eChallenges e-2008 Conference*.
39. Kapitza R, Schmidt H, Bartlang U, Hauck FJ. A generic infrastructure for decentralised dynamic loading of platform-specific code. *Distributed Applications and Interoperable Systems* (*Lecture Notes in Computer Science*, vol. 4531/2007), Indulska J, Raymond K (eds.). Springer: Berlin/Heidelberg, 2007; 323–336. *Proceedings of the 7th IFIP WG 6.1 International Conference*, *DAIS 2007*, Paphos, Cyprus, 6–8 June 2007.
40. Reiser HP, Bartlang U, Hauck FJ. A reconfigurable system architecture for consensus-based group communication. *Parallel and Distributed Computing and Systems* (*PDCS 2005*), Zheng SQ (ed.). ACTA Press, 2005; 680–686. *Proceedings of the 17th IASTED International Conference on Parallel and Distributed Computing and Systems PDCS*, Phoenix, AZ, 14–16 November 2005.
41. Lamport L. The part-time parliament. *ACM Transactions on Computer Systems* 1998; **16**(2):133–169.
42. Bartlang U, Müller JP. DhtFlex: A flexible approach to enable efficient atomic data management tailored for structured peer-to-peer overlays. *ICIW*, vol. 3, Mellouk A, Bi J, Ortiz G, Chiu DKW, Popescu M (eds.). IEEE Computer Society Press: Silver Spring, MD, 2008; 377–384. *Proceedings of the Third International Conference on Internet and Web Applications and Services*.
43. Jinyang, Stribling J, Gil TM, Morris R, Kaashoek MF. Comparing the performance of distributed hash tables under churn. *Peer-to-Peer Systems* (*Lecture Notes in Computer Science*, vol. 3). Springer: Berlin/Heidelberg, 2005; 87–99.
44. Gray J, Helland P, O'Neil P, Shasha D. The dangers of replication and a solution. *SIGMOD '96*: *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*. ACM: New York, NY, U.S.A., 1996; 173–182.
45. Howard JH, Kazar ML, Menees SG, Nichols DA, Satyanarayanan M, Sidebotham RN, West MJ. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems* 1988; **6**(1):51–81.
46. Levy E, Silberschatz A. Distributed file systems: Concepts and examples. *ACM Computing Surveys* 1990; **22**(4):321–374.
47. Gummadi KP, Saroiu S, Gribble SD. King: Estimating latency between arbitrary internet end hosts. *Proceedings of the SIGCOMM Internet Measurement Workshop* (*IMW 2002*), Marseille, France, 2002.

DOI: 10.1002/cpe