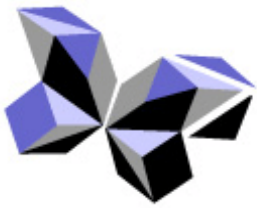

Digital Business Ecosystem

Contract n° 507953

Workpackage 24: DBE Implementation

Deliverable D24.7: Implementation of the Distributed Storage System

Project funded by the European Community under the "Information Society Technology" Programme

Contract Number: 507953

Project Acronym: DBE

Title: Digital Business Ecosystem

Deliverable N°: 24.7

Due dates: December 2006

Delivery Date: December 2006

Short Description:

The Distributed Storage System (DSS) is a Digital Business Ecosystem (DBE) infrastructural service designed to support the persistence of arbitrary content onto the network of DBE nodes. To realise this functionality a distributed storage layer, a DBE file-system layer and associated example and browser applications have been developed. This document describes the implementation of the Distributed Storage System in its entirety. The associated source-code is publicly available as part of the Swallow SourceForge project accessible online at http://www.sourceforge.net/projects/swallow.

Authors: John Kennedy, Robert Lee

Partners contributed: Intel Ireland Ltd.

Made available to: Public

Versioning

Version	Date	Author, Organisation
0.1	1 Dec 06	John Kennedy, Robert Lee, Intel Ireland Ltd. – Initial version
0.2	10 Jan 07	John Kennedy, Intel Ireland Ltd. – Reviewer Feedback, miscellaneous updates
0.3	29 Jan 07	John Kennedy, Intel Ireland Ltd. – Minor edits
1.0	7 Feb 07	John Kennedy, Intel Ireland Ltd. – Final approval of edits from internal reviewers

Quality check:

1st Internal Reviewer: Javier Noguera, SUN Microsystems

2nd Internal Reviewer: Chris Foley, Waterford Institute of Technology

Table of Contents

Table of Figures

Table of Tables

Table of Listings

1 Executive Summary

The Digital Business Ecosystem (DBE) Distributed Storage System (DSS) consists of an infrastructural service and associated software that allows arbitrary content to be persisted on the DBE network. The relevant software includes a fully distributed storage layer known as the DSS, a DBE file-system layer named the DBE File System (DFS), and various sample and client applications, including a DFS browser.

The software has been implemented following an analysis of potential use-cases and the current state-of-the-art in distributed storage systems. The DSS has been architected to fit smoothly within the DBE infrastructure, utilising other infrastructural services where appropriate. Where possible, the software has been designed to present the lowest overhead to the client in question, given the client's functional requirements. Functionality has been implemented using design patterns that facilitate the straight-forward swap-out of core components. Although the software is service-based, client-side helper classes have been developed to minimise the amount of code required to interact with the DSS.

Following a brief introduction into the architecture of the DBE, the requirements for the DSS are presented in terms of potential use-cases, functional requirements and environmental requirements. A summary of the state-of-the-art is provided, followed by the high level architecture and design of the required software. The implementation is then explained with reference to class diagrams for each component. Following a conclusion, the bibliography and an appendix of JavaDocs for the client-side classes are detailed.

This document is provided as an accompaniment to the DSS software. The associated source-code is publicly available as part of the Swallow SourceForge project accessible at http://www.sourceforge.net/projects/swallow.

2 Introduction

The Digital Business Ecosystem (DBE) [1] can be described as being a peer-to-peer semantically-aware service-oriented architecture (SOA) for Small and Medium sized Enterprises (SMEs). Architecturally [2], the DBE has been divided into an Execution Environment (ExE), a Service Factory and the Evolutionary Environment (EvE).

The ExE [3] includes the servent, which hosts services, as well as a variety of infrastructural services. The Service Factory is an Eclipse*-based development environment known as DBE Studio [4], and a distributed model-repository known as the Knowledge Base. The Service Factory is used to design, develop and deploy DBE services. The final component, the EvE [5], is designed to help optimise the running of the system, and is inspired by biological evolutionary processes.

All three of these blocks have extensive requirements for distributed functionality and these needs are met by the DBE's distributed infrastructural services. The Knowledge Base, for example, is an infrastructural service, as is the implementation of the EvE - realised as the Habitat service. The Distributed Storage System (DSS) is the infrastructural service designed to provide generic network-based storage facilities for the DBE.

This architecture is illustrated in Figure 1.

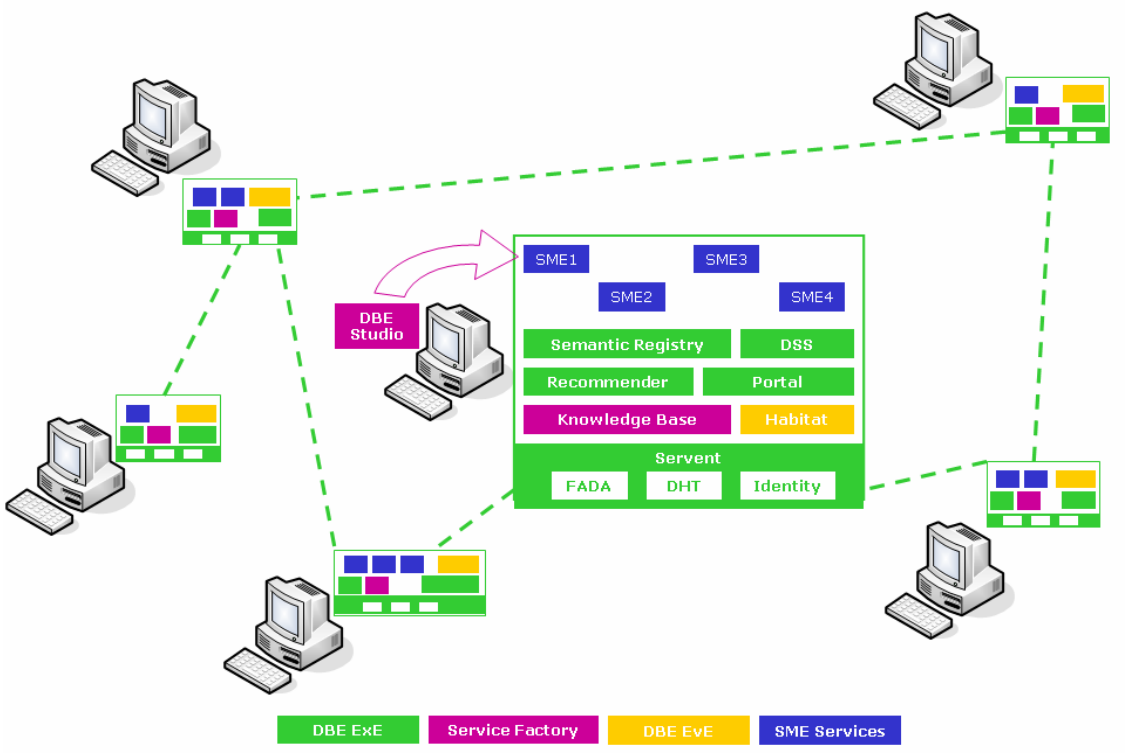

Figure 1: High Level Architecture of the DBE

* Other names and brands may be claimed as the property of others.

At its most fundamental level, the DSS must allow any DBE node to store content in the DSS, and once saved, any arbitrary DBE node on the network must be able to retrieve that content if required. More specific functionality, such as the ability to create "directories" and "files" with particular names, is often quoted as being storage functionality, but can in fact be considered to be a completely different layer of functionality that exists on top of an underlying storage system. This is analogous to the local storage systems of standard personal computers where the hard disk corresponds to the storage layer while the file-system (e.g. FAT) is an extra layer on top, typically implemented by the operating system. As file systems are particularly well known and useful storage facilities, implementing equivalent functionality for the DBE has been considered within scope for this project.

Envisioned as an infrastructural service on the DBE, the DSS has the opportunity to use and build on the other components of the DBE that have been realised. In particular, the DSS can interact with the servent – the software that hosts services – and other infrastructural services. Potentially useful infrastructural services include:

- Semantic Registry [6] – a distributed service directory
- FADA [7] – a peer-to-peer distributed registry of service-proxies
- DHT [8] – a peer-to-peer distributed hash-table
- Identity [8] – a distributed certificate store and identity manager

This document describes the implementation of the DBE's DSS. Although the software has developed over three years, this document will focus its description on the software available at the end of this project. In particular, the Java* and .NET* web-services-based hybrid implemented during the first phase of the project is not described.

Chapter 3 addresses the requirements of the system. Potential use-cases are described, functional requirements are extracted, and general environmental requirements are detailed. With this understanding of the problem, the existing state-of-the-art in distributed storage systems is explored in Chapter 4. This investigates various aspects of distributed storage systems, including physical storage, partitioning of content, indexing, querying, replication, trust and fairness.

The architecture and design of the DSS is then addressed in Chapter 5. The high-level architectural decisions are described, and the design of the two resulting layers of software, the storage system and file system, is discussed. Chapter 6 describes the implementation of the DSS. The details of the various DSS components are diagrammed and documented, and general deployment considerations are explained.

Chapter 7 provides a conclusion to the document, summarising the achievements and suggesting some potential future work. The Bibliography is followed by an Appendix including detailed JavaDocs describing the classes and interfaces that consumers of the DSS are most likely to use.

* Other names and brands may be claimed as the property of others.

3 Requirements

Although the high-level concept of the DBE was considerably advanced even early-on in the lifetime of the project, significant evolution of the technology and implementation has occurred since its inception. Rather than impose arbitrary fixed requirements on the DSS which may quickly have become stale, potential use-cases were used to help guide its design and implementation. After exploring potential use-cases with appropriate partners, realistic functional requirements were extracted. Additional requirements for the DSS followed-on from the fact that it would be used by and must be compatible with the DBE infrastructure itself. These use-cases, functional requirements and environmental implications are now described in some detail.

3.1 Potential Use-Cases

From a high-level point of view, there are a wide variety of potential clients for the DSS infrastructural service. Essentially, the list includes every component that could benefit from storing arbitrary content in a distributed and replicated environment. Conceivable classes of DSS clients are illustrated in Figure 2.

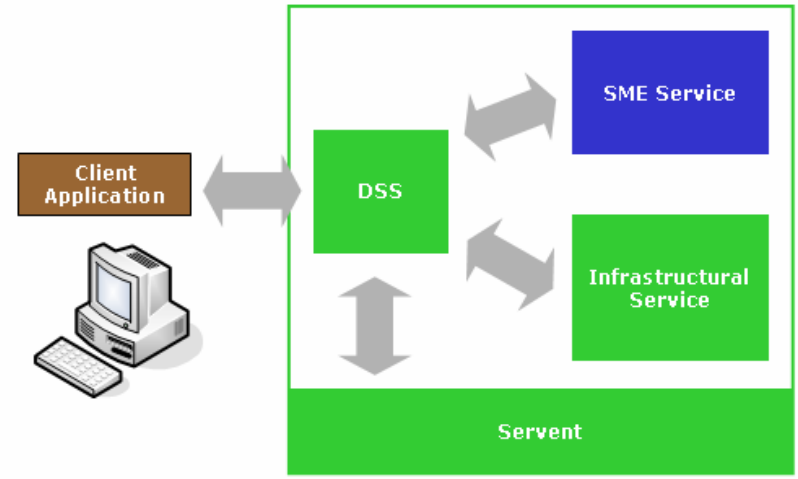

Figure 2: Potential clients of the DSS

In terms of examples, the core servent could use the DSS to persist logging or user profile information that might be useful to other servents or services. A metering infrastructural service could use the DSS to store raw metering data. An SME service could use the DSS to persist binary documents. A client application could use the DSS to store the application's session state.

Some specific use-cases identified during the course of the project are presented in Table 1 through to Table 5.

Use-Case 1	Metering
Objective	To persist the detailed metering data recorded by the metering filter upon arbitrary DBE service invocations. This data can later be retrieved by a mediation service, and ultimately be used to prepare a bill for the consumption of the service.
Actors	DBE Metering Service, DBE Mediation Service
Pre-Conditions	DBE Metering and DBE Mediation services are started and configured appropriately. DBE Metering services typically employ a DBE Mediation service that is hosted on a different servent.
Post-Conditions	n/a
Description	• The DBE Metering Service is informed of a service invocation by the local metering filter. The appropriate metering data is identified and extracted. • The DBE Metering Service persists this metering content on the DSS. The DSS returns an identifier for this content. • The DBE Metering Service informs the DBE Mediation Service of the identifier for the metering data. • The DBE Mediation Service retrieves the metering content from the DSS. The metering data is processed.
Storage Requirements	• The raw metering content is in arbitrary XML format. • Although the content will typically be accessed once (within minutes, days or weeks of storing) for legal reasons the metering content may need to be available for years. • The content is private, and should not be publicly accessible. • The content is used for billing purposes and so is highly critical: it is not acceptable for data to permanently disappear or become temporarily unavailable. • The data will not be updated. It is “write once, read many”.
Dependencies	DBE Metering Service DBE Mediation Service
More Information	DBE Deliverable D36.2, A set of Accounting Software Building Blocks [9]

Table 1: Use-Case 1 – Metering

Use-Case 2	Contracts
Objective	To persist formal contracts in an accessible place.
Actors	DBE Contract Editor
Pre-Conditions	DBE Contract Editor configured to point to a servent.
Post-Conditions	n/a
Description	• The DBE Contract Editor generates a contract document. • The DBE Contract Editor persists this contract on the DSS. The DSS returns an identifier for this content. • The user of the DBE Contract Editor informs any relevant party of the identifier for the contract. • Anyone wishing to retrieve and view the contract can retrieve it by supplying the appropriate identifier. • The contract is then displayed to the user.
Storage Requirements	• The contract content is in arbitrary format, typically pdf. • For legal reasons the contract may need to be available for years. • The contract is private, and should not be publicly accessible. • During negotiation it is critical that the contract is available. Once agreed, it is potentially acceptable for small periods of unavailability to be tolerated. • During negotiation the contract may change arbitrarily. Once agreed the contract will remain static - it will not be updated. Although this scenario could be considered to be “write many, read many”, if updates to contracts are considered to create replacement contracts then “write once, read many” could be more appropriate.
Dependencies	DBE Contract Editor
More Information	DBE Deliverable D32.6, DBE Model for Generic Level DBE Contracts [10]

Table 2: Use-Case 2 – Contracts

Use-Case 3	Local Service Pool
Objective	To persist the local service pool in an accessible, redundant place.
Actors	DBE Habitat Service
Pre-Conditions	DBE Habitat Service is configured appropriately
Post-Conditions	n/a
Description	• Over time the DBE Habitat Service creates and expands its Local Service Pool, a container of EveServices which are essentially pointers to DBE Services plus related information. • The DBE Habitat Service persists its Local Service Pool to the DSS and receives an identifier in return. • When the Local Service Pool needs to be retrieved, e.g. on Habitat Service startup, the Habitat Service requests the content from the DSS by supplying the appropriate identifier. • The DBE Habitat Service then resumes processing the Local Service Pool.
Storage Requirements	• The Local Service Pool is in an arbitrary format. • Due to the dynamic nature of Services on the DBE, the Local Service Pool would not need to be persisted for longer than several months. • The Local Service Pool content may be updated or replaced an arbitrary number of times, i.e. “write many, read many”.
Dependencies	DBE Habitat Service.
More Information	DBE Deliverable D9.4, EvE Service Implementation [11]

Table 3: Use-Case 3 – Local Service Pool

Use-Case 4	File System
Objective	To allow arbitrary directories to be created and arbitrary files to be stored in a network addressable space.
Actors	Any potential DBE client software
Pre-Conditions	Client software can access a DBE servent
Post-Conditions	n/a
Description	• The client can create directories, subdirectories and store and manipulate files in a network addressable space.
Storage Requirements	• Stored files can be retrieved, replaced, deleted, updated or appended to just like standard local files. • Standard file-system authentication, authorization and logging functionality is possible. • Arbitrary namespaces are supported. • Content is always available. • Long-term availability of content may be required. • "write many, read many" operations must be supported.
Dependencies	n/a
More Information	n/a

Table 4: Use-Case 4 – File System

Use-Case 5	User Profiling
Objective	To store user profiles defining user preferences
Actors	User Profile Editor. Potentially user analysis tools in the future.
Pre-Conditions	Users have an identity.
Post-Conditions	n/a
Description	• Alterations to a user profile are either manually or automatically defined. The User Profile editor requests the DSS to store the user profile. • At some time in the future, the user searches for a service on the DBE. • The Recommender requests the DSS to retrieve the appropriate user profile. • The appropriate user profile is used to help return the most appropriate results to the user.
Storage Requirements	• User profiles are in XML format. Binary content (e.g. graphical logos) may be associated with them in the future. • User profiles need to exist as part of the associated user account. To avoid stale data a limit of perhaps 6 months inactivity could be used to purge unused user profiles. • Unavailability of a user profile would result in potentially nonsensical query results returning to the user and would be unacceptable. • The contents of User Profiles need to be kept private. • User Profiles may be updated (replaced) regularly - perhaps many times a day - or may be rarely updated. This corresponds to “write many, read many”.
Dependencies	n/a
More Information	DBE Deliverable D7.2, Implementation of Profiling Mechanism [12]

Table 5: Use-Case 5 – User Profiling

3.2 Functional Requirements

From these potential use-cases, some high-level functional requirements were derived:

1. Content stored on the DSS would need to be available from any other node connected to the DBE.
2. Due to the arbitrary nature of the hosts on which the DSS would be deployed, replication of content would be required to provide for data redundancy. Different levels of redundancy may be appropriate for different use-cases.
3. Although it would need to be possible to encrypt content, encryption would not need to be mandatory.
4. Some content would need to be stored within an arbitrary namespace such as in a custom file system – but this would not always be required.
5. Some use-cases require standard File-System-like functionality to be available: the manipulation of folders/directories, subfolders/subdirectories, and files themselves.
6. The DSS would need to be able to store content of arbitrary size and format for an arbitrary amount of time.

3.3 Environmental Requirements

The DSS is part of, hosted by, and accessible through the DBE. This in itself introduces some environmental requirements that need to be accommodated.

Building on the core ethos of the DBE, the DSS must be Small and Medium sized Enterprise (SME) friendly. It must have a minimal cost to entry – not just in terms of zero up-front pricing but also in terms of time required to use and learn. As such it should employ well known open-source technologies where appropriate. It should operate on infrastructure that SMEs could be typically assumed to have in terms of hardware, software and network connectivity. Specifically, it should not force the purchase of any particular hardware, software or networking technologies.

As the DBE is designed to support 24-hour commerce and related activities across a peer-to-peer network with no centralised resources [2], this philosophy must be embraced by the DSS. Components should be distributed and without a single point of failure. This does not rule out, however, the prospect of arbitrary SMEs building their own value-added proprietary storage-service offerings on top of or compatible with the DSS for which appropriate fees may be charged. This model should also be supported by the DSS.

The system must also be self-maintaining – no central administration should be required. To facilitate continuous autonomous operation, some automatic facility to avoid build-up of stale content is required.

The DSS should be implemented as a DBE service to maximise its compatibility with the underlying infrastructure. From a technical point of view, this introduces some specific technical requirements. The software should be

developed in Java, so as to be able to run on multiple operating systems. Specifically, it should run on the same versions of the Java Virtual Machine as are supported by the servent. The software should also integrate with the servent and other DBE infrastructural services wherever appropriate to maximise efficiency and minimise duplication of functionality.

4 State of the Art

Armed with an understanding of the use-cases, functional requirements and environmental considerations concerning the DSS, it is appropriate to review the current state-of-the-art of distributed storage systems. A review of the literature concerning distributed storage systems is presented in this chapter.

Although numerous formal academic and industrial papers have been published in this field, an equally impressive community of software developers and enthusiasts is also very active in this area. These latter contributors are responsible for many of the internet-scale distributed storage systems that are in widespread use today. To attempt to reflect the significant experience gained by this community, the literature surveyed has included websites, blogs and online articles, in addition to the more traditional sources.

Rather than summarise the individual designs and implementations of numerous distributed storage systems, several key aspects of distributed storage systems have been identified and are discussed in some detail. References are made to specific implementations where appropriate.

The areas discussed include:

- Physical Storage
- Differentiating Storage Systems from File Systems
- Partitioning Content
- Indexing and Querying
- Replication
- Security & Trust
- Fairness

4.1 Physical Storage

For flexibility, cost and performance reasons, alternatives to hard-disk drive technology such as tape and CD are only used in specialist distributed storage systems where almost permanent persistence of typically very large files is the over-riding objective [13].

Within the enterprise, hard-disk based technologies such as Storage Area Networks (SANs) and Network Addressable Storage (NAS) are the physical infrastructure on which storage systems are frequently built. The former, SANs, allow for disk drives and drive controllers to be accessible on a network via a single computer (server). NAS's, on the other hand, allow for each of the disk drive units (for example) to receive an individual, dedicated, network address.

From a DBE point of view, however, SAN's require all the storage to be physically co-located (typically in the same server room), introducing a single point of failure and administration requirement. NAS's, on the other hand,

would impose potentially expensive hardware requirements on typically non-technical SMEs. For these reasons, both of these enterprise solutions are not considered relevant for the DBE DSS.

By far the most common solution for realising physical storage for distributed storage systems is to use the regular hard disks on the computers participating in the network. Initially popularised with the original Napster platform [14] (although not the existing Napster product/service), all client-side peer-to-peer distributed storage systems on the internet use this approach. These include FreeNet [15], BitTorrent [16], PAST [17], Gnutella [18], and the Cooperative File System [19], etc.

Some recent, very large scale distributed storage systems have moved to the concept of using stripped-down commodity personal computers (PCs) as the building blocks on which the system is based. Microsoft's BitVault [20] is one such example, referring to these stripped-down machines as Smart Bricks. The Google File System [21] also relies explicitly on commodity hardware components, and is designed to handle component failures as "the norm rather than the exception".

4.2 Differentiating Storage System from File System

Arguably the first widely popularised peer-to-peer distributed storage system, Napster was designed to share MP3 files alone. This implementation stored the content on individual peers, whilst the index was maintained at a central server. This separation of file indexing from storage mirrors exactly what happens in a standard computer: the hard-disk exposes storage functionality (in terms of blocks, cylinders and bytes) whilst the operating system implements the file system (in terms of file allocation tables, directories, file names, etc.).

Several distributed storage systems explicitly differentiate between their storage system and their file system. Frangipani [22], for example is a file system built on top of petal [23]. Others, such as IVY [24] and OceanStore [25], deliver pure storage system functionality only.

Pure storage systems typically refer to their content using the hashes of the content. This allows any peer to verify that the content has been neither maliciously nor inadvertently altered. OceanStore [25], IVY [24], and the Cooperative File System [19] all use this approach.

Some storage systems provide for immutability by using key-verified blocks to point to the blocks of hash-verified content. Other approaches have also been used to support the updating of content, including using distributed logs to manage updates as in IVY [24].

IVY is a decentralised peer-to-peer file system that, unusually, delivers traditional file-system semantics and is able to detect conflicts whereby multiple users are attempting to update the same content. The system state is stored in the set of all log files, one per participant, each stored in a DHash distributed hash table. Only the local log is

updated when modifications to content are made – whilst all logs are consulted whenever content needs to be retrieved.

Many storage systems, however, avoid the complications and overhead that locking requires and are optimised to support long-tem read-only archival of content…BitVault [20], Google File System [21] and BitTorrent [16] employ this approach.

4.3 Partitioning Content

In order to store potentially enormous content, many systems split content into blocks, which can be either of fixed or variable size. For example, the Cooperative File System [19], Pond [26] and Venti [27] all impose a maximum block size of 8 KB.

However, such systems can struggle when tasked with storing large volumes of data. A 1GB filesystem, for example, would require approximately 131,000 blocks to be located, retrieved and concatenated to be used. To reduce this overhead, Two-Level, Self-Verifying Data for peer-to-peer storage has been proposed [28]. It allows blocks to be stored within larger containers called extents. To locate particular content, the client must supply both the extent and block name. For internal management, caching, transport and querying, however, the storage system can operate at the extent level only.

A table of the blocks (including extents, if appropriate) that make up a particular content needs to be stored in the distributed storage system also. If immutability is required, then these indexing-blocks can not be hash-verified. Instead, key-verified techniques are often used to maintain confidence and control in the indexing blocks.

4.4 Indexing & Querying

In order to locate content in the peer-to-peer world several approaches to indexing and querying have been used.

The original Napster [14] used a central database to implement it's content index. Whilst ensuring that all content in the index can be retrieved, this approach can have scalability problems when a tremendous number of client nodes are participating in the system. The reliance of the system on a central source of data can also leave the system exposed. A tier of read-only database nodes that mirror the master database (which receives the index updates) is one way to overcome this problem, but introduces latency into the system.

The original Gnutella [18] used a different approach, that of a flooding algorithm. This attempted to locate particular content on any of its neighbours, or neighbour's neighbours, out to a predefined radius, or number of hops. This approach does not guarantee that content will ever be located, but is completely peer-to-peer and does not depend on a centralised infrastructure.

BitTorrent [16] has employed torrent files on web servers to dynamically track where copies of the content are currently available. This approach requires that the user knows the fixed address of the torrent for the content that they want to retrieve. As soon as they successfully retrieve the various blocks of the content, they can start to host those blocks and respond to other user queries for the content, thus distributing the load of the system.

Distributed Hash Tables (DHTs) is a relatively recent technique which revolves around each node maintaining the indexes for content in a particular range of the total namespace. As nodes join or leave the network the range of content for which each member is responsible can change dynamically and automatically. As each peer has a unique, typically random, node identifier that also belongs to the namespace, queries for location of a particular piece of content can be routed quickly, typically within O $\log_2$ N hops, where N is the number of nodes.

Typically peers that join a DHT are assigned completely random node identifiers, and so are connected to logical neighbours that physically may reside anywhere else on the network. This can give rise to situations where a query is passed from a peer A to a neighbour B and on to another peer C on the logical network, even though A and B might be on the same subnet and C might physically be in a completely different Autonomous System (AS). Adaptive Connection Establishment [29] has been proposed as a way of dynamically optimising the connections between neighbours to improve the efficiency of queries. This algorithm involves constructing an overlay multicast tree among the source and neighbours within a certain radius, and with optimising connections with neighbours outside this radius. Simulated improvements of 65% in query cost and 35% in response times have been demonstrated.

4.5 Replication

To provide for redundancy, many distributed storage systems replicate content including Google File System [21] and BitVault [20]. The partitioning of content has already been discussed, but when combined with replication some novel techniques have been developed. It has been determined that rather than subdivide data into a number of blocks, and replicate each of these blocks an appropriate number of times so that each is sufficiently replicated, it is possible to create erasure-coded blocks that include enough additional data to allow the entire content to be regenerated with just a subset of the blocks [31]. The same level of redundancy can be generated with less overall consumption of disk resources. Mojo Nation [31], and the open-source equivalent mnet [32], are examples of storage systems that adopt this approach.

In terms of managing replication in a decentralised system, Microsoft's BitVault [20] distributes this challenge. Built with three key design goals in mind: low Total Cost of Ownership, extremely high reliability and availability, and simplicity, BitVault employs a DHT index. However, rather than have the index control the replication, the replicas themselves have enough awareness and power to rebuild the index if necessary, and regenerate any replicas as required.

Some systems allow the amount of replication to be controlled by the storer of the content. PAST [17], for example, kept track of the storage consumed by each user, for which users were expected to pay. Users could specify how much replication they required, balancing redundancy with cost.

4.6 Security & Trust

Security, trust and related issues such as anonymity are tackled in a wide variety of ways in modern distributed storage systems. Freenet [15], for example, is a completely decentralised, anonymous, open-source peer-to-peer storage system whose primary objective is facilitating the freedom to publish any information without fear of censorship. Extensive precautions are made to ensure that content cannot be associated with a particular user.

All nodes on Freenet are equivalent. Each node keeps track of a number of logical neighbours, and how well they can accept or retrieve content. Each piece of content is assigned a key – that key is required to retrieve the content. Content can be split into multiple chunks, and all content is encrypted and replicated. Messaging follows a "Next Generation Routing" algorithm, which is somewhat similar to a Distributed Hash Table approach. A particular type of key – a Signed Subspace Key - can be used to verify that a particular pseudo-anonymous author created some content. A slightly less secure Keyword Signed Key (KSK) can be used to encrypt content using a human readable string, hence facilitating sharing of content to a restricted group of users.

Comparable to Freenet in its approach, Tor [33] is a follow-on project to FreeHaven [34], both sacrificing performance to focus on the goal of providing a network infrastructure on which content can be shared as anonymously as possible.

To preserve identity, on the other hand, some systems go to great length to track the owner of content, and the quantity of storage they have consumed. The PAST [17] distributed storage system, for example, was designed to use physical Smart Cards to implement security, and keep track of quotas associated with individual users.

4.7 Fairness

In an open, decentralised storage system the maintenance of fairness can be a challenge: what is to prevent a user from abusing the system? The Cooperative File System [19] is a read-only peer-to-peer storage system that uses a decentralised quota system to reduce the possibility of abuse. It is based on the IP address of the peer attempting to store content on the system – each IP address may only use a certain percentage of disk space on the peer in question.

Mojo Nation [31] employs the concept of a digital currency (whose denomination is known as mojo) to encourage user to contribute resources to the network. Mojos are consumed when a user stores or queries the storage system. Mojos are earned when resources are contributed to the system. A trusted third party tracks transactions and balances. This mechanism has the added benefit of encouraging load-balancing: on a busy server, the queries with higher "bids" are serviced first, encouraging all but the richest users to use less loaded nodes.

5 Architecture and Design

Based on the potential use-cases gleaned from potential consumers of the DSS, a study of the state-of-the-art, and practical experience in implementing a centralised corporate peer-to-peer content distribution system, it was possible to make several key architectural and design decisions.

At a fundamental level, the DSS was going to be a DBE infrastructural service and so would be Java based, and rely on the Servent and other infrastructural services for application hosting and peer-to-peer network functionality.

At a functional level, it was decided that rather than implement just a file-system interface, other lower-level interfaces should also be supported. Specifically, the software was split into a pure Distributed Storage System: the DSS, and a DBE File System: the DFS. Applications not requiring file-system functionality could then interact with the DSS directly, avoiding the overhead and associated performance costs that a file system would impose.

From an implementation point of view, it was appreciated that many alternative techniques existed for many aspects of the software including indexing, storage and encryption techniques. Rather than mandate particular approaches, it was decided to partition the software appropriately (for example using the Factory pattern [35]) to support the dynamic swapping of components, without having to recompile and redeploy the entire solution.

Although DBE infrastructural services by their nature provide a stateless interface, this interface cannot be invoked without first receiving a proxy to the service and then invoking the appropriate methods. Rather than force all client applications to implement custom code to perform these operations, it was decided to provide client helper classes to wrap this functionality up in more-easy-to-use, potentially stateful, client-side classes. These client classes could expose the functionality using potentially familiar interfaces, reducing learning overhead for developers.

The main components of the resulting high level design are illustrated in Figure 3.

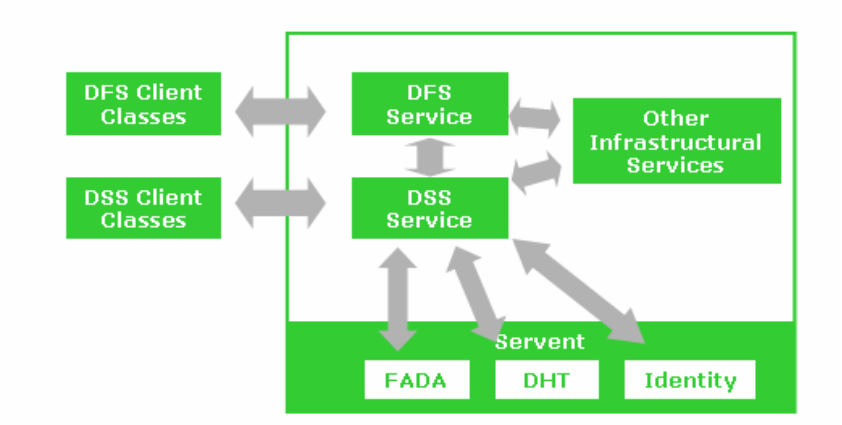

Figure 3: High Level Design of the DSS

5.1 DSS Design

It was decided to implement the DSS-specific software across a collection of components. Each component would consist of a collection of classes and possibly interfaces, and would be packed as a Java Archive – a JAR file. This would allow distribution of the DSS client to only require the bundle of DSS client and common JAR files, while the DSS infrastructural service would require only service-specific (and common) Jar files.

The DSS client classes were designed to provide a straight-forward and easy to use interface to the DSS service. Two basic operations would be supported: the writing of arbitrary content, which would result in an identifier being returned to the invoker, and the retrieving of content previously stored. Rather than implement a DSS-specific security layer, helper methods would be provided to assist client users to encrypt their content using the standard Java encryption facilities.

To allow for autonomous operations (and self-clean-up), a time-to-live must be set for all content stored on the DSS. Background processes in the DSS service automatically purge content from the DSS once the time-to-live has expired. Clients can extend the time-to-live of content at any stage if required.

Content would be identified by automatically generated hashcodes. In the highly unlikely event of collisions, identifiers would include an additional post-fix to uniquely identify it. Basing identifiers on hash-codes allows the DSS services to verify content integrity after content transfer.

To allow content of arbitrary length be stored, content of excessive size would be subdivided into blocks. A block listing the identifiers of the constituent blocks would also be persisted, and the identifier of this meta-block would be used as the identifier for the content as a whole. Initially, sub-blocks would be simple partitions of the content, but the possibility of using erasure-coding in future iterations would be factored into interfaces.

To provide for redundancy, blocks would be replicated across multiple nodes. As absolute guarantees cannot be made of availability, the degree of redundancy required can be specified by the storer of the content. Although higher degrees of redundancy will, for example, increase the number replications, they will take more time and bandwidth to persist and so the storer will have to balance redundancy versus performance.

Regarding indexing, the decentralised nature of the DBE requires a decentralised index and a DHT was deemed most appropriate. With the DSS requirements factored into the peer-to-peer design of the DBE [8], it was appropriate to build on top of the DHT implementation by Trinity College Dublin. This would necessitate a dss-index core component to overlay the default DHT core-component implementation. The DSS service would then use this overlay for indexing purposes. As the DHT implementation was not due for completion until the end of the project, it was foreseen that alternative indexes would be required in the interim and so a factory-type model would be employed to instantiate the index component to be used. This would allow for replacement with alternative implementations of the index in the future without the need for recompilation.

In order to be self-maintaining, the system would need to be able to accommodate the fact that blocks may become unavailable on the nodes on which they originally resided. This would happen if a node was switched off, for example. To handle this eventuality, each node would insert its entries into the DHT index with a limited time-to-live of, perhaps, one week. This index time-to-live would be renewed for another week, before the week expires. This renewal process would continue until the user-specified time-to-live of the content has expired. In this way, stale entries would be automatically purged from the DHT within a relatively short timeframe.

To maintain redundancy given the churn of nodes, a background process on each DSS service would review the status of each of the blocks stored locally. The DSS Index would be checked to see where the copies of each block existed. If the number of copies fell below the required amount, attempts would be made to replicate copies onto an appropriate number of arbitrary DSS nodes.

In terms of persistence of content, this would be done by storing blocks in a suitable directory on the local file-system. As future implementations may appreciate the ability to store blocks as blobs in relation databases, or on alternative media such as local CD or tape facilities, the storage component was also designed using the factory model to facilitate its dynamic replacement if required.

The organisation of the DSS-specific components is illustrated in Figure 4.

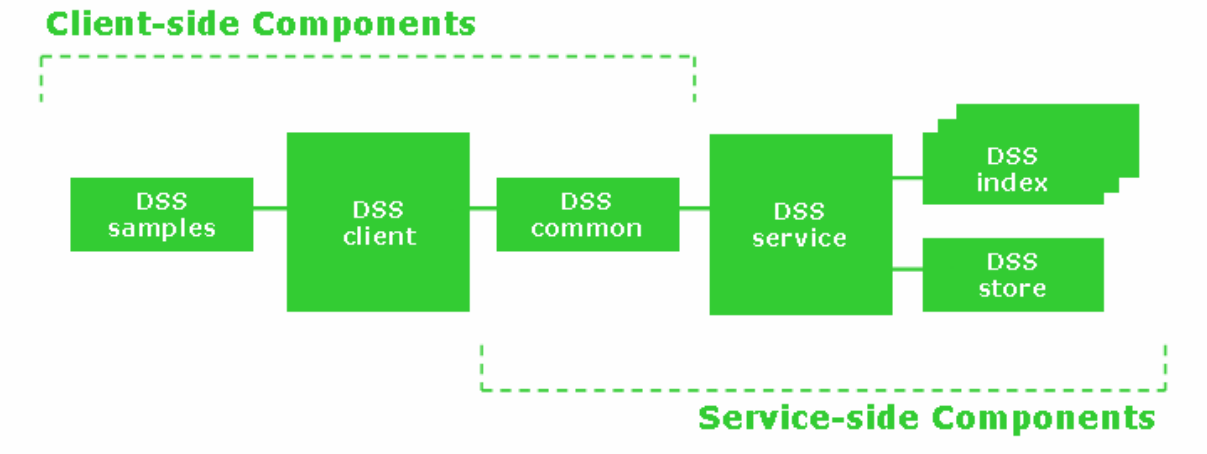

Figure 4: Organisation of the DSS-specific components

The DSS samples component referred to in Figure 4 was provided to demonstrate how the DSS could be used in various scenarios. A simple “Hello World” application stores a simple note onto the DSS, and then retrieves it. A simple “Hello World Service” illustrates how a DBE service could persist content onto the DSS.

5.2 DFS Specifics

To satisfy the more specific requirements of some of the use-cases, a DBE File System (DFS) was also designed as part of this project.

The DFS would share many key design characteristics with the DSS. It too would be partitioned across multiple components to simplify distribution and minimise package-size. While the content within files would be persisted on the DSS, the table that mapped directory and file names to content identifiers on the DSS would need to be stored in a DFS index. This functionality would be separated from the DFS service and the factory model adopted to facilitate the use of alternative DFS index implementations.

To present a familiar interface to Java developers, client-side helper classes inspired by those in java.io would be used. In particular, DFS files and directories would be manipulated using methods as close to those of java.io.File as possible. Some differences would be necessary, however, and will now be described.

Based on an analysis of standard file-systems on common operating systems, it was decided that DFS files and directories would be subject to the following rules. A name of a DFS file or directory would be able to contain any printable Unicode characters except for the following which would be forbidden: " * / : ; < > ? \ |. Individual names longer than 250 characters in length would not be permitted. Names containing only a sequence of the period (‘.’) or space (‘ ‘) characters, or ending with either of these characters, would likewise be forbidden. Adoption of these rules would maximise the amount of range of file and directory names that could be manipulated in the DFS.

In terms of a separator between directory names, and directory and files, it was decided to allow '\', '\\', '/', and '//' to all be valid DFS separators. However, client helper methods that would return a path would use the default separator of the local file system. DFS paths would begin with a separator by default, but this would not be mandatory. DFS file paths would not be case sensitive, but would maintain case.

As the DFS would be implemented following the service model, state would not be readily preserved, and so no concept of “Current Directory” would exist. Therefore only absolute paths, rather than relative ones, would be supported by the DFS.

To avoid arbitrary build-up of content over time, DFS files would have a time-to-live associated with them. Directories, as they could contain arbitrarily deep trees of sub-directories and files, would not have a time-to-live associated with them to avoid surprise deletions of trees of content. Files would be automatically deleted once their time-to-live had expired.

To improve the performance of client applications, additional methods to those in java.io.File would be created where relevant to minimise the number of network calls required to perform typical file-system operations.

Regarding the index of the DFS, it was appreciated following the state-of-the-art analysis and after considerable exploration of potential options that a distributed consistent file system with usable performance would not be feasible to implement. Mindful of the need to avoid single points-of-failure which could hold SMEs ransom so-to-speak, the concept of each SME being responsible for their own file system or systems was entertained.

Given that SMEs would typically have their own DBE servent that hosted the services that allowed interactions with their business; it was considered reasonable that their node would be available whenever they wanted to be accessible on the DBE. Thus, a DFS file system service that they might host would also be available whenever they were on the network. This would correspond with the occasions when they might want to have access to a file system, and so the idea of any SME who wanted file system functionality creating their own file system service instance was adopted.

Although this would introduce a single-point-of-failure in terms of the index for their file system, the file system would still be distributed, accessible from any node on the DBE. Furthermore, the possibility of technical SMEs offering alternative value-added instances of DFS would be opened up. This corresponds well to the standard file-system model, whereby some file systems are optimised for speed, some for security, some for redundancy and robustness etc., each having their own advantages and disadvantages. Similar opportunities would be possible on the DBE also.

To realise this approach, a generic interface into a DFS was defined in such a way that particular DFS instances could be accessed. A basic implementation of this DFS would be realised as part of the project.

It was decided that the reference DFS implementation would persist the content of files to the DSS, and maintain its index mapping DSS identifiers to DFS files and directories inside a PostgreSQL database [36]. The enterprise-class support, reliability, redundancy options and performance of PostgreSQL as well as the uncomplicated open-source licensing model it employs were the key drivers for choosing this relational database management system. To maximise integrity, a normalised set of tables would be defined with as tightly defined constraints as possible, and all access to these tables would be limited to an appropriate set of SQL functions. These functions would have all the necessary logic within them to prevent incomplete or corrupt data from entering the file system index.

The DFS service implementation would invoke the appropriate SQL functions in order to write into or read from the index.

To implement time-to-live, the relevant SQL functions would ignore all entries whose time-to-live had expired, whilst a dedicated maintenance task would periodically purge the expired entries from the DFS index. The DSS would automatically manage the purging of the actual content independently.

The resulting organisation of the DFS-specific components is illustrated in Figure 5.

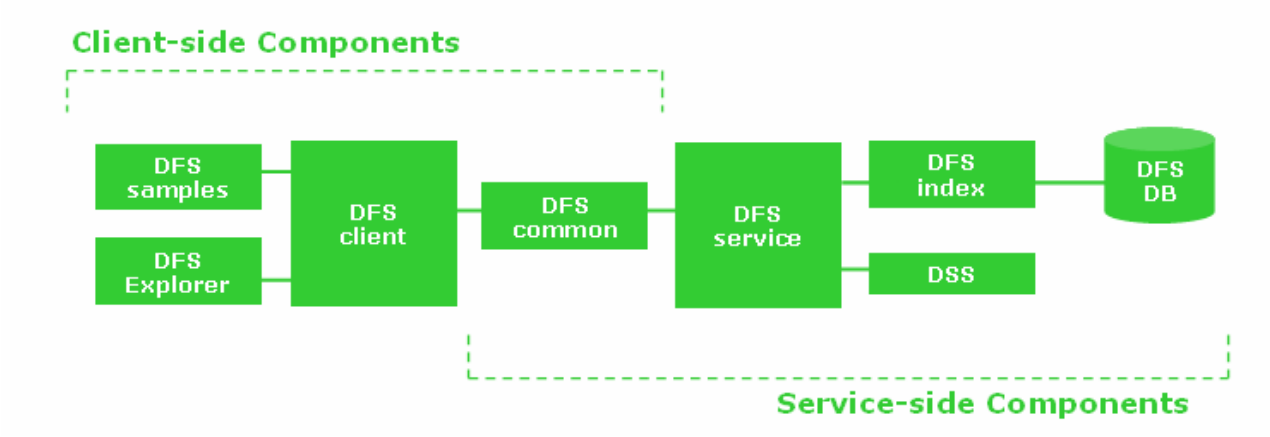

Figure 5: Organisation of the DFS-specific components

The DFS samples and DFS explorer components referred to in Figure 5 are provided to illustrate how the DFS could be used. A simple “Hello World” application stores a simple file onto an arbitrary DFS instance, and then retrieves it. The DFS Explorer is a fully-functional, graphical, browser application.

6 Implementation

6.1 Introduction

This chapter describes the implementation of the DSS and DFS. The various components are first listed, then described individually with the help of appropriate UML diagrams, database schemas and figures. Rather than descend into minute interface details here, detailed descriptions of the Application Programming Interface (API) for all of the public components can be found in Appendix A (DSS) and Appendix B (DFS). The configuration and file structure required to deploy the software onto the servent is also explained.

6.2 DSS Implementation

The DSS-specific functionality has been implemented via the components listed in Table 6.

Table 6: DSS-specific components

Component	Description
dss-client	Client-side utility classes
dss-common	Classes and indexes that are required on both client-side and service-side
dss-service	Service implementation and internal interfaces
dss-store-localdir	Implementation of store interface persisting to the local file system
dss-index-local	Implementation of index interface persisting to the local file system
dss-index-postgresql	Implementation of index interface persisting to a (possibly remote) PostgreSQL database
dss-index-service	Implementation of index interface persisting to a (possibly remote) dedicated DSS Index service
dss-index-dht	Implementation of index interface persisting to the DSS-index DSS overlay
dss-samples	Examples demonstrating how to use the DSS

The implementation detail of the individual components is now described in some detail. This is followed by a description of some more generic implementation details that may be of interest.

6.2.1 dss-client

The dss-client component includes classes that can be used by clients of the DSS to simplify their interactions with the DSS. The package and classes of the dss-client component are illustrated in Figure 6.

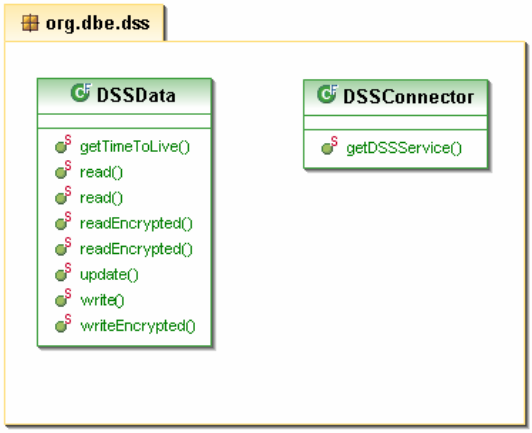

Figure 6: Class Diagram for dss-client

The **DSSData** class is the primary client class, and allows the various read(), write() and update() methods to be invoked in a static fashion. Additional helper methods are provided to facilitate the encryption of data. The content is encrypted (or decrypted) using the supplied algorithm and key.

The **DSSConnector** class is provided to manage connections to the DSS service. It removes the need for a proxy to be located and a connection to be formed for every individual call to the DSS service. This class is typically not invoked by client applications – the methods in the DSSData class invoke it automatically as required.

The static nature of these classes allows the DSS to be invoked by making direct calls to DSSData.read() and DSSData.write() – an explicit instantiation of the DSSData class not being required.

6.2.2 dss-common

The dss-common component contains elements that are required on both the client-side and service-side of the DSS implementation. The packages, class, interface and methods of the dss-common component are illustrated in Figure 7.

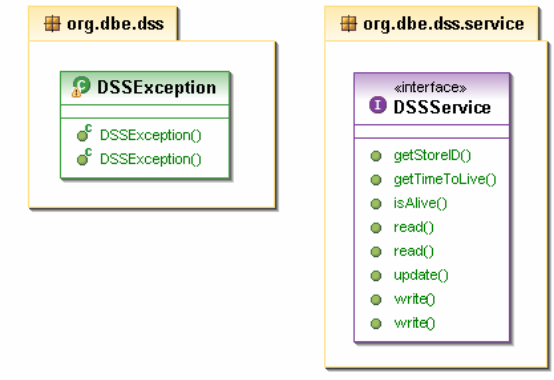

Figure 7: Class Diagram for dss-common

The **DSSException** class is a class designed to encapsulate a DSS-specific exception. It extends the well known java.io.IOException class. If an exceptional condition is detected anywhere in the DSS source-code that cannot be described by an existing well-known Java Exception, then a DSSException is populated with an appropriate message and thrown to the calling code.

The **DSSService** interface is an interface for the DSS infrastructural service. The stateless methods allow content to be stored or retrieved, and the time-to-live to be updated or returned. By default, content is replicated synchronously, but for performance optimisation purposes this can be overridden via the overloaded write() method. An overloaded read() method allows a specific subset of the content to be retrieved – the subset is specified via an offset and length parameter. This can be useful when only a certain number of bytes of a large piece of content are required: only the required bytes are sent from the service over the network to the client, thus speeding up the response potentially significantly. The interface also allows the time-to-live of content to be explicitly queried and updated. The isAlive() method allows the availability of the service itself to be quickly determined.

6.2.3 dss-service

The dss-service component contains the core implementation of the dss-service, and also defines interfaces for service elements that are not included in this component. The packages, classes, interfaces and methods of the dss-service component are illustrated in Figure 8.

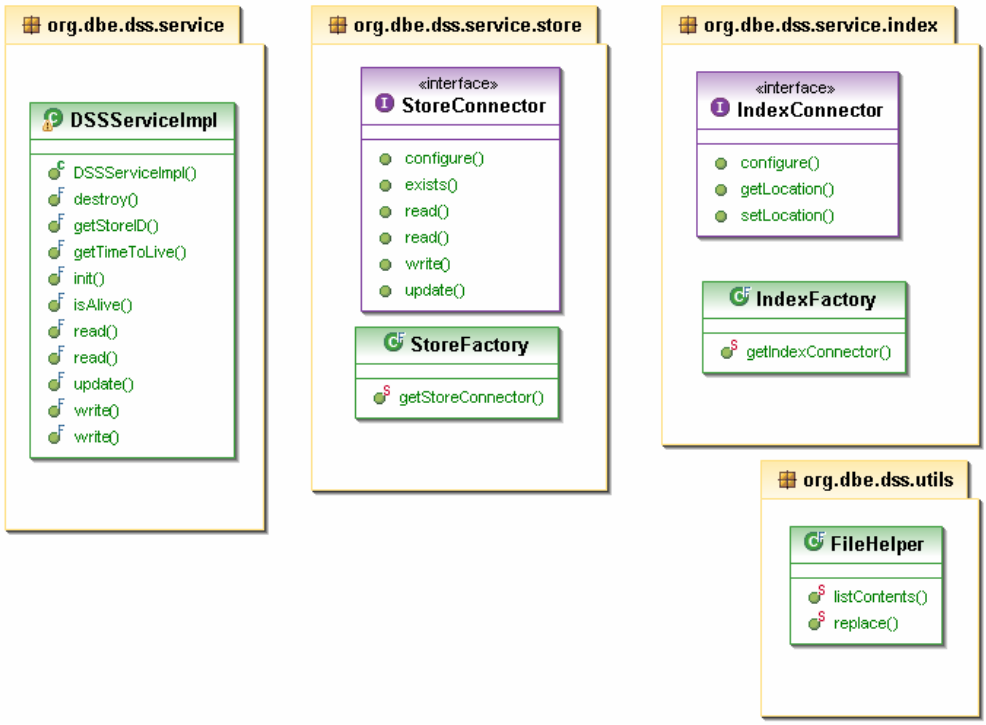

Figure 8: Class Diagram for dss-service

The **DSSServiceImpl** class is the core implementation class for the DSS service. It implements the DSSService and servent Adapter interfaces. On service initialisation, the class checks to see if the service has already been

configured with a unique store ID. If not, one is generated and saved into the local configuration file, deployment.xml. This unique ID allows the location of content to be stored in the index. This ID does not change over time. Connections to the local index and local store are also created at initialisation, based on service configuration settings.

The read() and write() methods implement the core storage algorithms. For writing, the content is stored locally (after splitting into blocks if required) and then these blocks are replicated, if required, onto arbitrary other DSS nodes an appropriate number of times depending on the parameters supplied. Meta-data about the content is included in each block. This metadata includes the time-to-live and replication requirements of the content, allowing each node to self-manage the blocks in the local store without the overhead of making network calls. The ID of each block is calculated by the write() operation. It is based on the hashcode of the content. If a hashcode collision is detected, a previously unused number is appended to the identifier.

For reading from the DSS, if the content is not already on the local node then it is requested from the nodes that have it according to the DSS index. The content is pulled locally from the node able to transfer the content the quickest and the requests to the other nodes are cancelled. If the block is a meta-block that contains a list of other block identifiers, then this process is repeated until all required blocks are retrieved. Note that if only a subset of the content is required, as specified by an offset and length, then only those blocks that contain this content are pulled locally. If a block is stored locally during the read process, then this new location is stored on the DSS index.

The **StoreConnector** interface is an interface to a separate component that implements a store for DSS content. This was done to allow storage mechanisms be replaced easily in the future. The basic read(), write(), exists() and update() methods are provided – note that update() only allows certain metadata (e.g. time-to-live) to be updated. The configure() method is provided to allow the store receive configuration parameters from the servent. This allows things such as the store ID and the local absolute directory of the service to be determined.

The **StoreFactory** class provides a connection to a store, instantiating the store based on configuration parameters. Initially the valid types of store were hard-coded into the factory class, but to ensure maximum flexibility the actual name of the class implementing the store is now completely parameterized itself. This allows arbitrary stores to be created and used in the future without having to recompile and redeploy the DSS service component itself.

The **IndexConnector** interface is an interface to a separate component that implements an index for DSS content. This was done to allow index mechanisms be replaced easily. The basic operations of setLocation() and getLocation() are supported. If the time-to-live of the entry in the index needs to be updated, the setLocation() method can be called. The configure() method is provided to allow the index receive configuration parameters from the servent, as some index implementations may need to know the absolute path of the service directory, for example.

The **IndexFactory** class provides a connection to an index, instantiating the index based on configuration parameters. Initially the valid types of index were hard-coded into the factory class, but to ensure maximum flexibility the actual name of the class implementing the index is now completely parameterized itself. This allows arbitrary indexes to be used without having to recompile and redeploy the DSS service component itself.

The **FileHelper** class is implemented to facilitate some standard file operations that service classes may find useful. These include replacing content inside files, and listing the contents of a directory.

6.2.4 dss-index-local

The dss-index-local component contains an implementation of a DSS index which just indexes the content stored locally. It was developed for testing purposes to facilitate development on other aspects of the system. The package, class and methods of the dss-index-local component are illustrated in Figure 9.

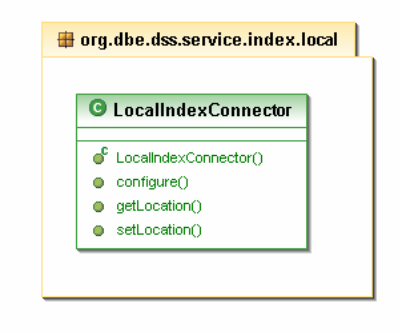

Figure 9: Class Diagram for dss-index-local

The **LocalIndexConnector** class implements the IndexConnector interface. Upon initialisation, a memory-resident hashtable is populated with the list of all blocks stored locally. This hashtable is appended to or searched by calls to setLocation() and getLocation() respectively. This implementation is not designed to be scalable, and does not even support a distributed DSS – it was developed purely for testing purposes on a single-node DSS.

6.2.5 dss-index-postgresql

The dss-index-postgresql component contains an implementation of a DSS index which persists the index in a PostgreSQL [36] database. It was developed to allow a functional, distributed DSS to be tested in advance of a distributed index being available. The package, class and methods of the dss-index-postgresql component are illustrated in Figure 10.

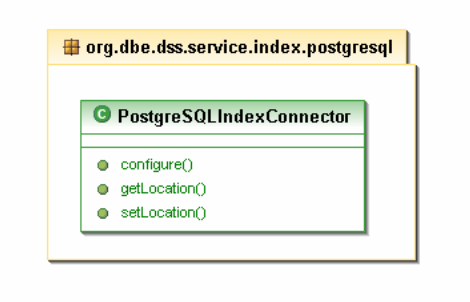

Figure 10: Class Diagram for dss-index-postgresql

The **PostgreSQLIndexConnector** class implements the IndexConnector interface. The setLocation() and getLocation() methods invoke dedicated functions in a PostgreSQL database which in turn manipulate a single table, store_blocks. The schema of the database is detailed in Table 7, and the functions are listed in Table 8. A composite primary key is defined across the store_id and block_id columns to allow multiple locations be assigned for each block. "NOT NULL" and "DEFAULT" constraints are used to minimise the possibility of invalid data being inserted into the database.

Table 7: Database Schema of Optional DSS Index

Table	*Field*	*DataType*	*Description*
store_blocks	store_id	varchar(50)	The ID of the DSS store at which this block is stored
	block_id	varchar(50)	The unique block ID
	expiry_date_time	timestamp	The date and time at which this content can be removed from the index
	load_date_time	timestamp	The date and time at which this entry was added to the index

Table 8: SQL Functions of Optional DSS Index

Function	*Description*
dss_writestoreblock	Assigns the specified location to the specified block with the specified time-to-live.
dss_getstores	Returns a list of stores at which the supplied block exists.

Although this optional DSS index supports a distributed DSS, and even though PostgreSQL is an enterprise-scale relational database management system, this approach nonetheless includes a single point of failure as it relies on a centralised index. This component was developed purely for testing purposes on a multiple-node DSS: the multiple DSS nodes are configured to communicate to a shared DSS index database.

6.2.6 dss-index-service

The dss-index-service component contains an implementation of a DSS index which persists the index in a dedicated dss-index service. It was developed to allow a functional, distributed DSS to be tested in environments where database connections could not be supported for network security reasons. The package, classes, interface and methods of the dss-index-service component are illustrated in Figure 11.

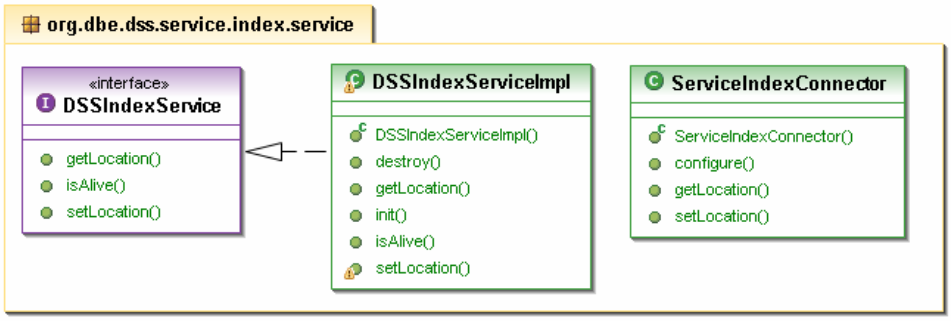

Figure 11: Class Diagram for dss-index-service

The **ServiceIndexConnector** class implements the IndexConnector interface. The setLocation() and getLocation() methods invoke servent calls which seek out a DSS Index Service and invoke corresponding methods. The dss node can be configured to search for and connect to a particular DSS Index Service in case multiple DSS Index Service exist on the network.

The **DSSIndexService** interface implements the servent Adapter interface, and describes the interface to the DSS Index service.

The **DSSIndexServiceImpl** class implements the DSS Index service. It is based on the dss-index-local component described previously, the major difference being that the hashtable is saved locally after every call of the setLocation() method.

This implementation was developed purely for testing purposes on a multiple-node DSS where database calls across network boundaries are not permitted.

6.2.7 dss-index-dht

The dss-index-dht component contains an implementation of a DSS index which persists the index onto a DBE Distributed Hash Table (DHT). The package, class and methods of the dss-index-dht component are illustrated in Figure 12.

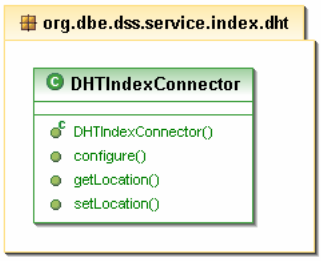

Figure 12: Class Diagram for dss-index-dht

The **DHTIndexConnector** class implements the IndexConnector interface. The setLocation() and getLocation() methods invoke dedicated functions in a DSS Index core component that overlays the DHT. The DSS Index core component exposes only that functionality of the DHT required for the DSS: the ability to write locations and time-to-live for block-identifiers and the ability to retrieve locations of particular block-identifiers. As the maximum time-to-live would be one week, the DHT would automatically purge obsolete entries within a matter of days.

6.2.8 dss-store-localdir

The dss-store-localdir component contains an implementation of a DSS store which persists the content onto the local file system. The package, class and methods of the dss-store-localdir component are illustrated in Figure 13.

Figure 13: Class Diagram for dss-store-localdir

The **LocalDirStoreConnector** class implements the StoreConnector interface. The write() method persists a block on the local file system, while the read() methods retrieve a block. An overloaded read() method allows a particular subset of a block to be retrieved. An update() method allows the time-to-live to be manipulated. By default, the blocks are stored in the home directory of the service; however, the store can be configured to persist in a completely arbitrary directory.

A separate thread is spawned at initialisation-time to manage the store. This thread oversees replication of locally stored content, and automatically removes content whose time-to-live has expired.

6.2.9 dss-samples

The dss-samples component includes sample applications that demonstrate the use of the DSS. The packages, classes, interface and methods of the dss-samples component are illustrated in Figure 14.

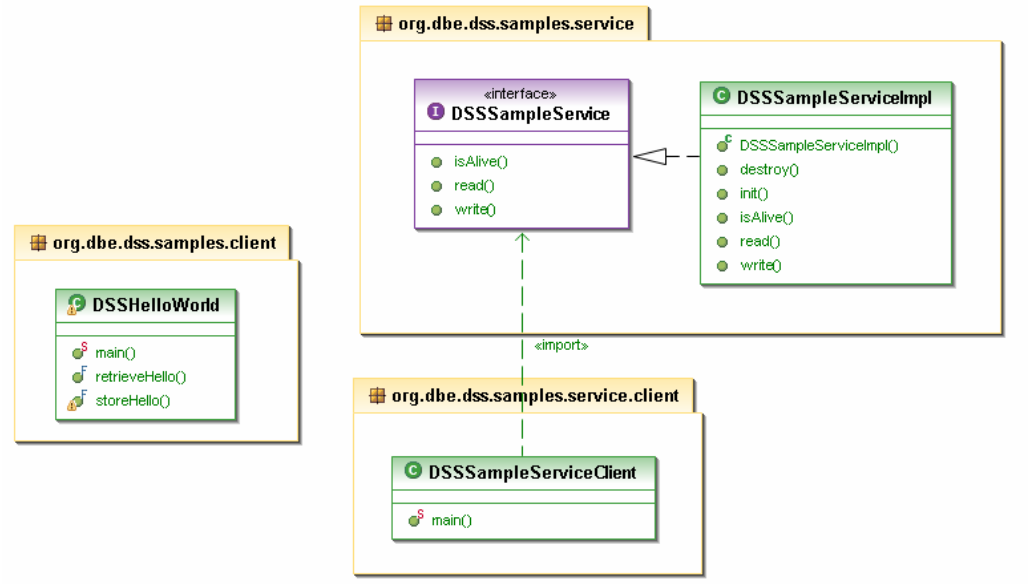

Figure 14: Class Diagram for dss-samples

The **DSSHelloWorld** example is a trivial example that generates some text (include the current date and time), persists it to the DSS using the DSSData class, and then retrieves it, also using the DSSData class.

The example in the org.dbe.dss.samples.service package demonstrates how the DSS can be invoked from a DBE service, but this time illustrating how to invoke the DSS service directly, without using DSSData. The service is defined by the **DSSSampleService** interface, and implemented by the **DSSSampleServiceImpl** class. The service allows the text that was supplied to be written, with the current date and time prepended to the text. The **DSSSampleServiceClient** class is a client application that invokes the DBE service.

The structure of the DSS samples is illustrated in Figure 15.

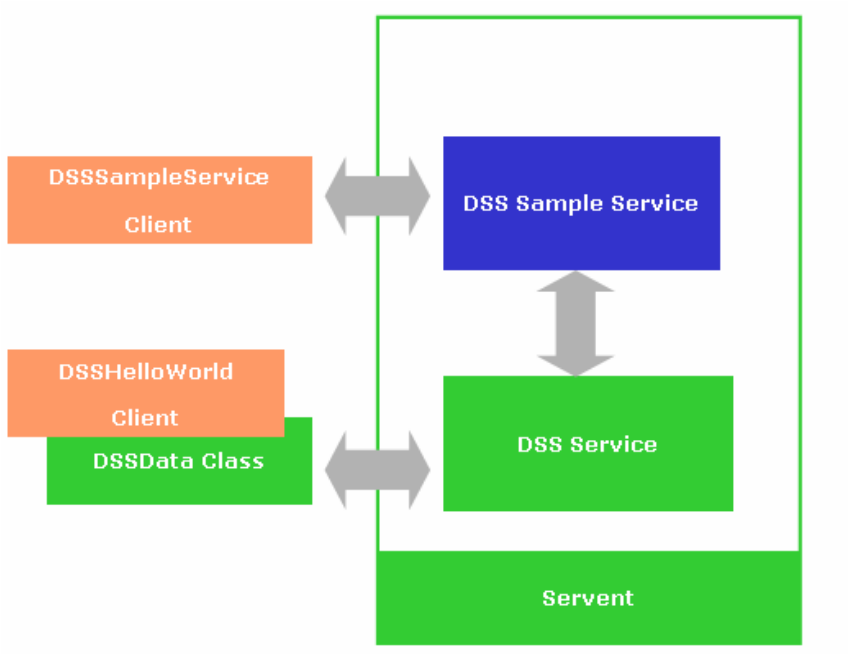

Figure 15: DSS Samples Structure

The samples have been instrumented with significant logging. The edited output of the DSSHelloWorld client is reproduced in Listing 1.

```
Calling storeHello()
printing line...
[DSS] 2007-01-14 23:44:45,574 DEBUG [main] DSSData - write(...,10000)
[DSS] 2007-01-14 23:44:45,574 DEBUG [main] DSSData - getDSSService()
[DSS] 2007-01-14 23:44:45,594 DEBUG [main] DSSConnector - getDSSService()
[DSS] 2007-01-14 23:44:45,594 DEBUG [main] DSSConnector - getNewConnection()
<snip>
writeHello() complete.
Calling retrieveHello()
[DSS] 2007-01-14 23:44:52,985 DEBUG [main] DSSData - read(nDKu+sAka57K07V6K+qCBVOwrlE=)
[DSS] 2007-01-14 23:44:52,985 DEBUG [main] DSSData - getDSSService()
[DSS] 2007-01-14 23:44:52,985 DEBUG [main] DSSConnector - getDSSService()
<snip>
The message read from the dss was:
2007-01-14 23:44:42:760 - Hello Distributed Storage System!!
...here is another line(!)
...printing complete.
done!
```

Listing 1: Hello World!

6.3 DSS Deployment

To deploy the DSS, the various component jar files are placed into appropriate directories and the appropriate settings are saved in the DSS service configuration file. An example of the required directory structure is provided in Listing 2. "<servent>" refers to the installation directory of the servent.

```
<servent>\deploy\DSSService\lib\dss-common-0.5.0.jar
<servent>\deploy\DSSService\lib\dss-index-service-0.5.0.jar
<servent>\deploy\DSSService\lib\dss-service-0.5.0.jar
<servent>\deploy\DSSService\lib\dss-store-localdir-0.5.0.jar
<servent>\deploy\DSSService\store
<servent>\deploy\DSSService\deployment.xml
```

Listing 2: DSS Service Directory Structure

The lib directory is where the service jar files are stored. In the example given, the service is using the DSS service index as implemented in the dss-index-service component. If the DSS service was using a different indexing mechanism, this file (dss-index-service-0.5.0.jar) would be replaced with the alternative indexing component.

The store directory is where the blocks are stored by the dss-store-localdir component. Each block is stored as one file. The name of each file is the ID of the block, typically the hashcode of the block (unless a hashcode collision has occurred).

The deployment.xml file contains the configuration settings for the service. An example deployment.xml file is reproduced in Appendix C. Note that comments begin with the characters <!-- and end with -->. The service includes logic to calculate a globally unique DSS Store ID if one is not configured. Regarding indexing, the valid settings are listed in Table 9, along with the corresponding jar file that must be present in the libs directory.

Table 9: DSS Indexing Configuration Options

Indexing Component	dss.index.type Setting	Required Jar file
dss-index-local	org.dbe.dss.service.index.local.LocalIndexConnector	dss-index-local-0.5.0.jar
dss-index-postgresql	org.dbe.dss.service.index.postgresql.PostgreSQLIndexConnector	dss-index-postgresql-0.5.0.jar
dss-index-service	org.dbe.dss.service.index.service.ServiceIndexConnector	dss-index-service-0.5.0.jar
dss-index-dht	org.dbe.dss.service.index..dht.DHTIndexConnector	dss-index-dht-0.5.0.jar

To help chose the appropriate DSS indexing solution for a given situation, a table summarising the main differences between the various DSS indexing components is provided in Table 10.

Table 10: DSS Indexing Comparison

Indexing Component	Supports Distributed Nodes	Redundancy Options	Decentralised Index	Comments
dss-index-local	No	No	No	For single node testing only
dss-index-postgresql	Yes	Yes	No	For multiple nodes, ideally on the same LAN
dss-index-service	Yes	No	No	For multiple node testing
dss-index-dht	Yes	Yes	Yes	For fully distributed DBE deployment

To assist with debugging and troubleshooting, the logic throughout the DSS service has been instrumented with detailed logging. This is implemented using the open-source Apache log4j framework [37]. The detail-level and destination for the DSS-specific logging information can be specified in the log4j.props file in the <servent> directory. The extract of this file in Listing 3 shows how to configure DEBUG level messages to be output into a rolling file in the <servent>\dss\log directory.

```
# DSS
log4j.logger.org.dbe.dss=DEBUG,dss
log4j.appender.dss=org.apache.log4j.RollingFileAppender
log4j.appender.dss.File=./dss/log/dss.log
log4j.appender.dss.MaxFileSize=1MB
```

Listing 3: DSS Log4j configuration

6.4 DFS Implementation

The DFS-specific functionality has been implemented via the components listed in Table 11.

Table 11: DFS-specific components

Component	Description
dfs-client	Client-side utility classes
dfs-common	Classes and indexes that are required on both client-side and service-side
dfs-service	Service implementation and internal interfaces
dfs-index-postgresql	Implementation of index interface persisting to a (possibly remote) PostgreSQL database
dfs-samples	Examples demonstrating how to use the DFS

The implementation detail of these components is now described in some detail.

6.4.1 dfs-client

The dfs-client component includes classes that can be used by clients of the DFS to simplify their interactions with the DFS. The package and classes of the dfs-client component are illustrated in Figure 16.

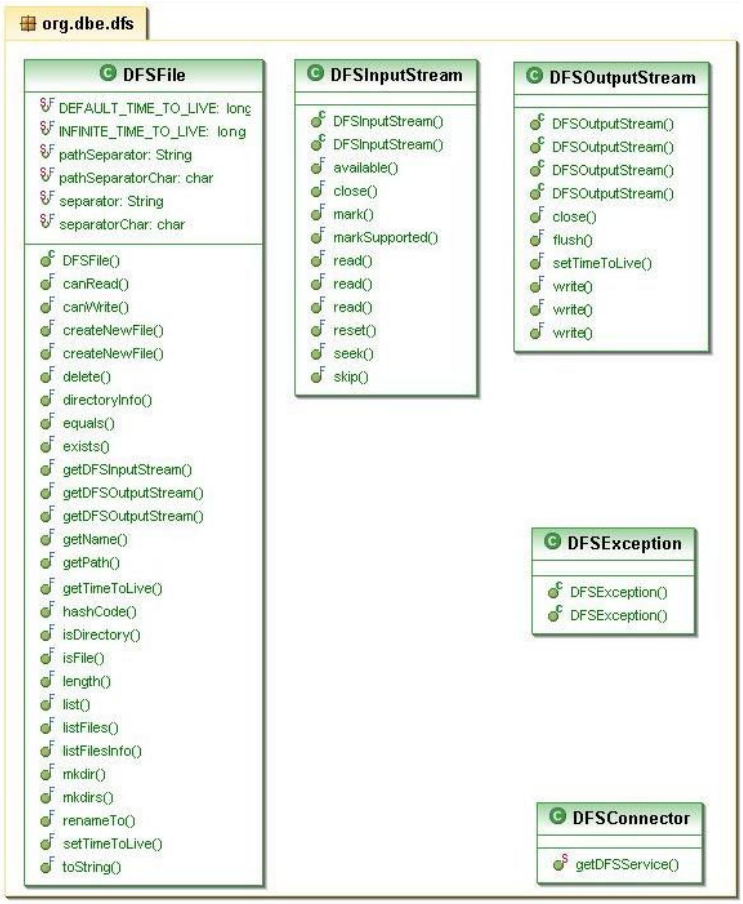

Figure 16: Class Diagram for dfs-client

The **DFSFile** class is an abstract representation of a DFS file or directory. The DFSFile methods are based on the methods of the well-known java.io.File class, but some DFS-specifics have been implemented as described in Section 5.2.

To elaborate on some of those specifics, the performance of client applications is improved by the implementation of the listFilesInfo() method. This returns the complete meta-data for all files in the specified DFS directory with just one service call. A directoryInfo() method has similarly been implemented to allow summary information about a directory to be retrieved in a solitary call. These methods are particularly useful for browser-type applications. To facilitate users of the DFS that may want to quickly create an InputStream or OutputStream into or out of a DFS file, the getDFSInputStream() and getDFSOutputStream() methods have been implemented.

Most methods in DFSFile are relatively simple wrappers over the corresponding DFS service methods. In some instances, however, the overhead of service calls can be avoided, as is the case with the getName() method, for example. The name of a file can be determined from the absolute path without having to communicate with the DFS service.

Although the strings "/", "//", "\" and "\\" are all supported as separators by the methods that have a path as an argument, all DFS client methods that return a path will return paths that use the default separator character of the local operating system.

The **DFSInputStream** class extends the java.io.InputStream class. The DFSInput class is used to read data from DFSFile objects. The read methods return an array of bytes and if an error occurs, a DFSException is thrown. The DFSInputStream implementation supports the optional InputStream "mark" functionality, allowing content to be marked by client applications and reread if required.

The **DFSOutputStream** class extends the java.io.OutputStream class and is used to store data into a DFSFile object. The data can replace the content in an existing DFSFile, or it can be appended to the end of the existing content. A portion of the data can also be written to the DFSFile based on the offset and length specified. If the length of the portion is much smaller than the entire set of data supplied, then just the subset is transferred. The DFSOutputStream throws a DFSException if an exception occurs.

The **DFSException** class is designed to encapsulate a DFS-specific exception. It extends the well known java.io.IOException class.

The **DFSConnector** class is provided to manage connections to the DFS service. It removes the need for a proxy to be located and a connection to be formed for every individual call to the DFS service. This class is typically not invoked by client applications – the methods in the DFSFile, DFSInputStream and DFSOutputStream classes invoke it automatically as required.

6.4.2 dfs-common

The dfs-common component includes classes that are used on both the client-side and service-side of the DFS implementation. The packages, interface and classes of the dfs-common component are illustrated in Figure 17.

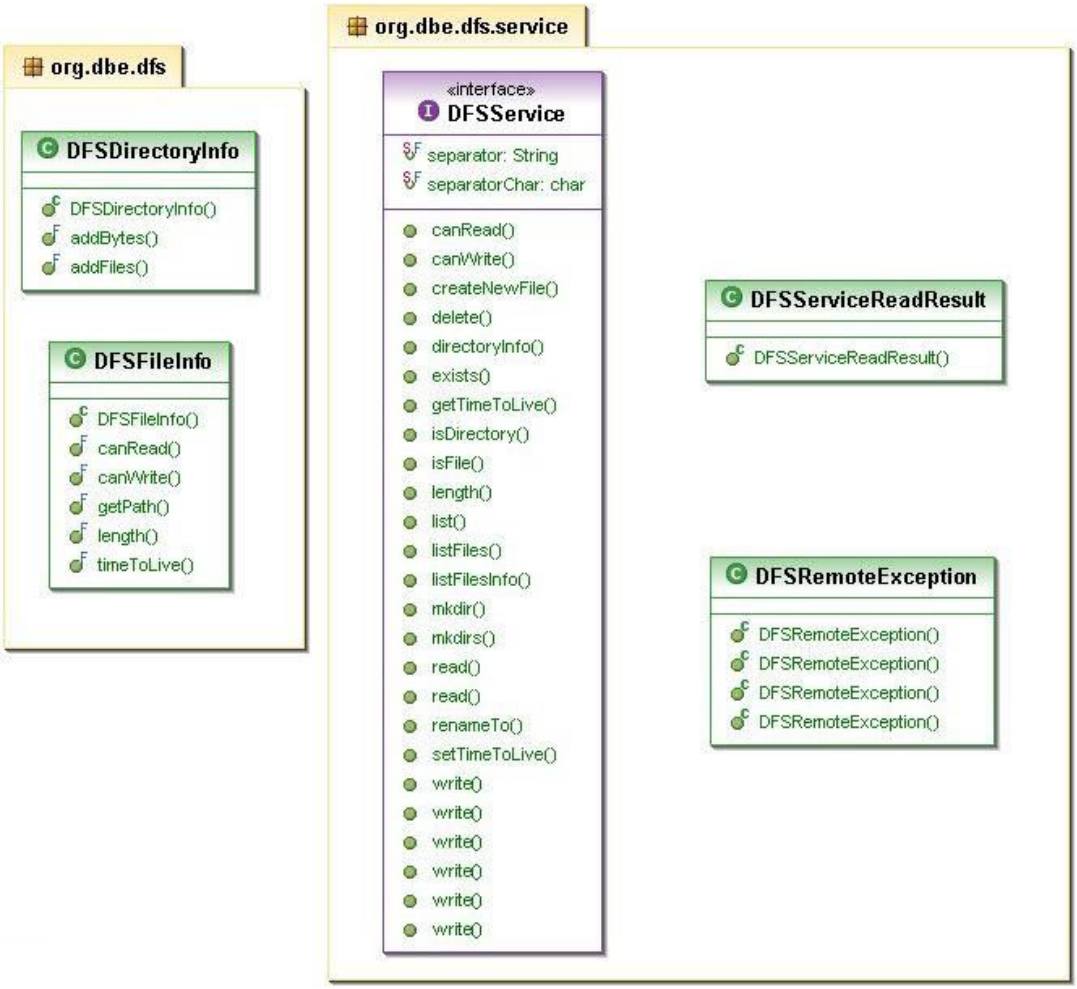

Figure 17: Class Diagram for dfs-common

The **DFSService** interface is an interface for the DFS service. The stateless methods allow files and directories to be manipulated. Overloaded methods for read() and write() allow portions of content to be retrieved or written. Other write() methods support content in byte array or integer form, and facilitate appending to or replacement of existing content. Although the strings "/", "//", "\" and "\\" are all supported as separators, the interface defines that the default separator character *returned* by the DFS *service* is "/".

The **DFSServiceReadResult** class is used to supply the result of a read operation. Both the bytes themselves and the length of the bytes read are returned. This facility allows the client to determine if end-of-file has been reached.

The **DFSRemoteException** class extends the Java Exception class. The exception is thrown if any DFS specific exception occurs.

The **DFSFileInfo** class contains information on individual DFSFile objects in the DFS. File attributes such as the length of the file, the time-to-live, and boolean values declaring if the file can be read and/or written to are encapsulated by this class.

The **DFSDirectoryInfo** class contains information on the total number of DFSFile objects contained in a directory and its sub-directories. The number of files in a directory, and its sub-directories, is stored, along with the size, in bytes, of the directory and its sub-directories.

6.4.3 dfs-service

The dfs-service component contains the core implementation of a basic, reference DFS service. An interface is also defined to facilitate the swapping-out of potential indexing implementations. The packages, interfaces and classes of the dfs-service component are illustrated in Figure 18.

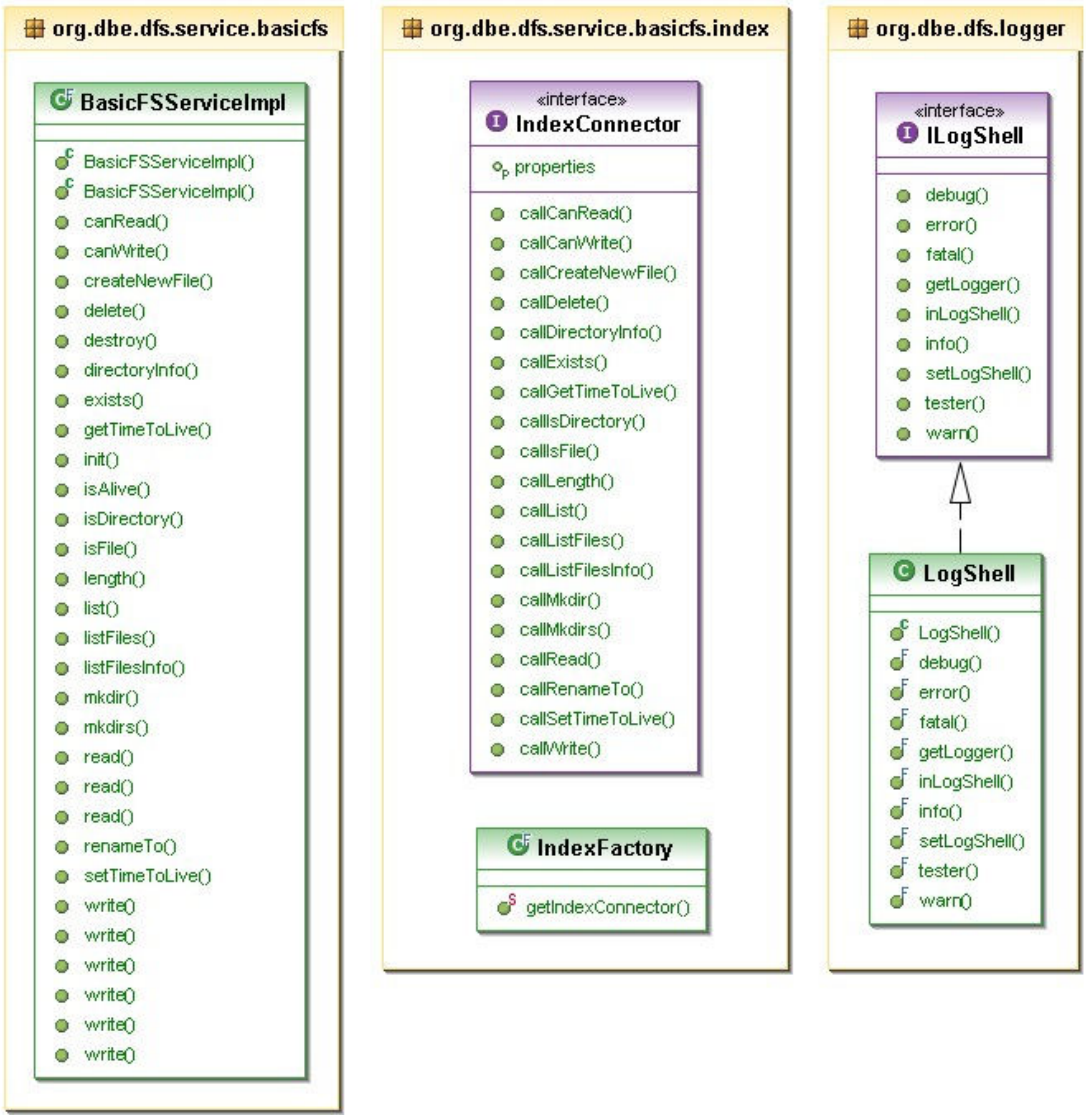

Figure 18: Class Diagram for dfs-service

The dfs-service component contains the interface to connect to a data source/repository. The index connector objects are created using the factory method design pattern. The factory method is a creational pattern that returns one of several types of subclass from an abstract base class depending on the data it receives. IndexConnector objects can be generated to communicate with different data sources depending on what is connected to the node.

The **BasicFSServiceImpl** class implements a basic reference DFS that persists its content on the DSS and maintains the mappings of directories, files and DSS-identifiers in a DFS Index. The index is not included in this component, instead its interface and a factory class are defined.

Most methods just make calls to the DFS index. The read() and write() methods are the main exception, as they interact with the DSS, as well as querying or updating the DFS index respectively.

The **IndexConnector** interface is an interface to a separate component that implements an index for DFS content. This was done to allow index mechanisms be replaced easily. Enough operations to support a basic file system are supported. Methods like read() and write() are not relevant for this interface as the content is not stored in the index.

The **IndexFactory** class provides a connection to an index, instantiating the index based on configuration parameters. Initially the valid types of index were hard-coded into the factory class, but to ensure maximum flexibility the actual name of the class implementing the index is now completely parameterized itself.

ILogShell is an interface for the LogShell class, and is implemented to allow the LogShell class to be mocked for unit-testing purposes. An interface is required for all classes that are mocked in java. Mocked classes are used to reliably simulate external classes whose functionality is not being tested in unit tests.

LogShell is a wrapper class on top of log4j [37], the third-party logging framework used by the project.

6.4.4 dfs-index-postgresql

The dfs-index-postgresql component contains an implementation of a DFS index which persists its content in a PostgreSQL database [36]. The package and class of the dfs-index-postgresql component are illustrated in Figure 19.

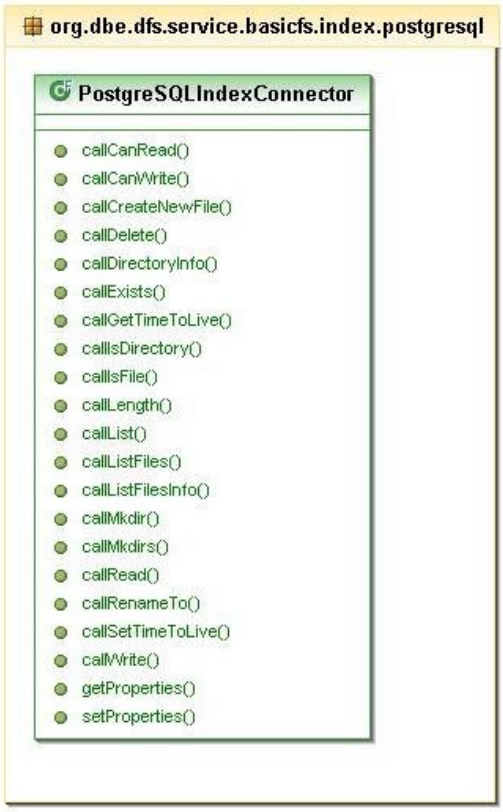

Figure 19: Class Diagram for dfs-index-postgresql

The PostgreSQLIndexConnector implements the DFS IndexConnector interface. Most methods are wrapper methods to equivalent SQL functions implemented in the PostgreSQL database for this reference DFS implementation.

The SQL functions are written to encapsulate as much functionality as close to the database tables as possible. Input parameters are checked for their validity before database updates are made. The two tables that enable this implementation are files and directories. Referential integrity and database constraints are used where possible to prevent the entry or removal of corrupting data. The database schema is listed in Table 12, and the SQL functions in Table 13.

Table 12: Database Schema of Reference DFS

Table	*Field*	*DataType*	*Description*
Files	file_id	big serial	The internal ID for this file
	directory_id	int (8)	The internal ID for the directory containing this file
	name	varchar (250)	The name of the file
	hash_code	varchar (50)	The ID of the content of this file stored on the DSS
	length	int (8)	The length of the content, in bytes
	expiry_date_time	timestamp	The date and time at which this file can be removed from the system
Directories	directory_id	big serial	The internal ID for this directory
	parent_id	int (8)	The internal ID for the directory containing this directory
	name	varchar (250)	The name of this directory

Table 13: SQL Functions of Reference DFS

Function	*Description*
bfs_canread	Returns whether this file or directory can be read. By default if the file or directory exists then this is true.
bfs_canwrite	Returns whether this file can be overwritten or appended to or if contents can be placed in this directory. By default if the file or directory exists then this is true.
bfs_createnewfile	If the path and time-to-live are valid and if the file does not already exist then creates a new file with this path.
bfs_delete	Deletes the file if it exists, otherwise if the path refers to a directory the directory is deleted if it is empty.
bfs_delete_old_files	Purges all files from the database whose time-to-live has expired.
bfs_directoryinfo	Returns information about the supplied directory if it is a valid directory path. The information is gathered by the internal function bfs_get_directory_info.
bfs_exists	Returns whether this file or directory exists.

bfs_get_directory_id	Internal function that returns the id of the directory corresponding to the name and parent directory id supplied.
bfs_get_directory_info	Internal function that returns the number of objects and size of these contents in the directory id supplied.
bfs_get_file_id	Internal function that returns the id of the file corresponding to the name and parent directory id supplied.
bfs_get_parent_id	Internal function that returns the id of the parent directory for the path supplied, assuming the path is not the root directory itself. If it is the root directory, NULL is returned.
bfs_get_path_type	Internal function that returns 1 if the path supplied corresponds to a directory which exists, 2 if it corresponds to a file which exists, and NULL otherwise.
bfs_gettimetolive	Returns -1 if the path supplied corresponds to a directory that exists, the number of seconds remaining for the path supplied if it corresponds to a file that exists, and -2 otherwise.
bfs_is_valid_path	Internal function that returns True if the path supplied is valid, otherwise false. Invalid paths do not begin with a separator, include two separators with no other characters between them, or end with a separator.
bfs_isdirectory	Returns whether the path supplied is a valid directory name, irrespective of whether it exists or not.
bfs_isfile	Returns whether the path supplied is a valid file name, irrespective of whether it exists or not.
bfs_length	Returns the size of the file corresponding to the path supplied if it is a file that exists, otherwise -1.
bfs_list	Returns a list of the files and directories contained in the path supplied, if it corresponds to a directory that exists. Otherwise NULL is returned.
bfs_listfiles	Returns a list of the files and directories contained in the path supplied using full and absolute paths, if it corresponds to a directory that exists. Otherwise NULL is returned.
bfs_listfilesinfo	Returns a list of the files and directories contained in the path supplied using full and absolute paths, along with additional information such as length, read only, is directory, is file etc., if it corresponds to a directory that exists. Otherwise NULL is returned.
bfs_mkdir	Creates a new directory corresponding to the path supplied if it is a valid path, the parent directory exists, and a file with the same name

	as the directory does not already exist.
bfs_mkdirs	Creates new directories and parent-directories corresponding to the path supplied if it is a valid path, and a file with the same name as the directory or any of the previously non-existing parent-directories does not already exist.
bfs_read	Returns the id of the block on the DSS that contains the content of the path supplied, if the path supplied corresponds to a file with content.
bfs_renameto	Renames the file or directory corresponding to the path supplied to the new path supplied, assuming that neither path is the root directory, both paths are valid, both paths have the same parent directory, and a file or directory does not all ready exists with the same name as the new directory or file being created, respectively.
bfs_settimetolive	Updates the time-to-live of the file corresponding to the path supplied, assuming it is a file which already exists.
bfs_write	Updates the database with the file size and id of the DSS block that corresponds to the content of the file being written to, assuming the path supplied is an existing file.

6.4.5 dfs-samples

The dfs-samples component includes a sample application that demonstrates the use of the DFS. The package and class of the dfs-samples component are illustrated in Figure 20.

Figure 20: Class Diagram for dfs-samples

The **DFSHelloWorld** class demonstrates how the classes in the DFS can be used. A “hello world” file is created on the DFS using DFSFile and then filled with some content using DFSOutputStream. The content includes the current date and time. The file is then retrieved from the DFS using DFSInputStream, and the contents displayed on screen.

6.4.6 DFS Explorer

The DFS Explorer is a JAVA SWING client application that presents a visual interface into an arbitrary DFS instance. The directories can be navigated and manipulated, and files (or entire directory trees) uploaded and downloaded. The class diagram for DFS Explorer is illustrated in Figure 21.

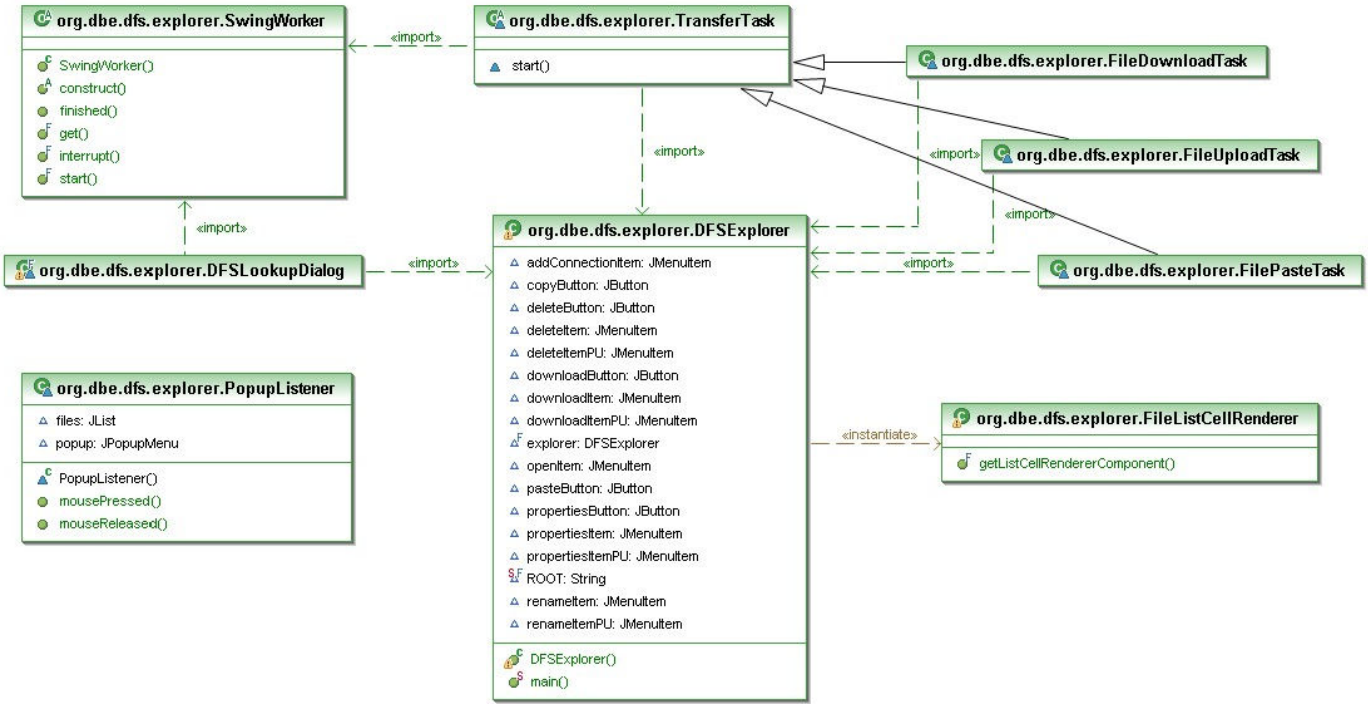

Figure 21: Class Diagram for DFS Explorer

The **DFSExplorer** class is the main class that defines user interface, and listeners for all the potential interactions.

The **DFSLookupDialog** class controls the connection to the DFS instance, locking the display while a connection is being made.

The **TransferTask** class is a base class that supports time consuming transfers being performed in a dedicated thread.

The **FileDownloadTask**, **FileUploadTask** and **FilePasteTask** classes all instantiate new threads to control the download, upload or paste of files into the requested places.

The **PopupListener** class handles popup menu events for the DFSExplorer.

The **SwingWorker** class is an abstract class that allows graphical operations to be performed in a separate thread, preventing the UI from locking up.

The **FileListCellRenderer** class is a class that is used to draw items in the file list. In the DFSExplorer, each item has an icon (e.g. folder icon) followed by a piece of text.

A screen shot of the DFS Explorer is illustrated in Figure 22.

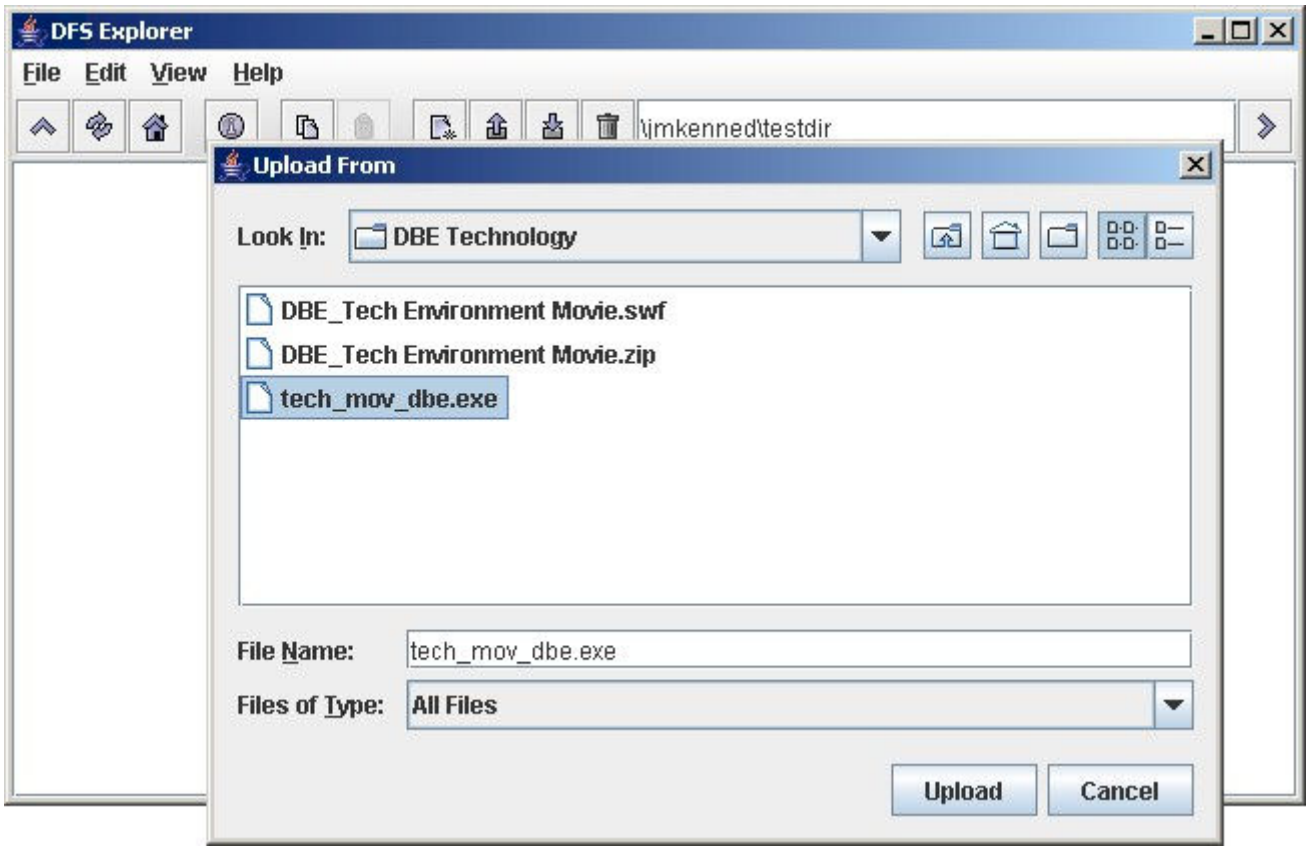

Figure 22: Screenshot of DFS Explorer

6.5 DFS Deployment

To deploy the DFS, the various jar files are placed into appropriate directories and the appropriate settings are saved in the DFS service configuration file. The required directory structure is explained in Listing 4. "<servent>" refers to the installation directory of the servent.

```
<servent>\deploy\DFSService\lib\dfs-common-0.4.0.jar
<servent>\deploy\DFSService\lib\dfs-index-postgresql-0.4.0.jar
<servent>\deploy\DFSService\lib\dfs-service-0.4.0.jar
<servent>\deploy\DFSService\lib\dss-common-0.5.0.jar
<servent>\deploy\DFSService\lib\postgresql-8.1-407.jdbc3.jar
<servent>\deploy\DFSService\deployment.xml
```

Listing 4: DFS Service Directory Structure

The database for the DFS index needs to be installed on an accessible PostgreSQL database. As many Internet firewalls restrict incoming traffic to HTTP ports, typically the PostgreSQL database will need to be on the same

LAN as the DFS service. To simplify the installation, batch files have been prepared which can create an empty DFS index database, create the appropriate security group and user accounts, create the table-schema, types and functions, and populate the database with appropriate initial configuration information. These scripts can be found in the \dfs\index-postgresql\bin directory on sourceforge.

The deployment.xml file contains the configuration settings for the service. The DFS configuration file is reproduced in Appendix D. Note that comments begin with the characters <!-- and end with -->. Each DFS service should be configured with a unique instance id to allow it to be explicitly referred to. The DFS index database location (url), username and password also need to be specified.

To assist with debugging and troubleshooting, the logic throughout the DFS service has been instrumented with detailed logging. This is implemented using the open-source Apache log4j framework [37]. The detail-level and destination for the DFS-specific logging information can be specified in the log4j.props file in the <servent> directory. The extract from this file shown in Listing 5 demonstrates how to configure INFO level messages to be output into the default servent log output location. In this file, lines that are commented out begin with the # character.

```
# DFS
log4j.logger.org.dbe.dfs=INFO,dfs
#log4j.appender.dfs=org.apache.log4j.RollingFileAppender
#log4j.appender.dfs.File=./dfs/log/dfs.log
#log4j.appender.dfs.MaxFileSize=1MB
#log4j.appender.dfs.layout=org.apache.log4j.PatternLayout
#log4j.appender.dfs.layout.ConversionPattern=%d %-5p [%t] %c{1} - %m%n
```

Listing 5: DFS Log4j configuration

7 Conclusion

This document has described the implementation of a fully distributed storage system and related software that has been developed specifically for the DBE. Potential use-cases, functional requirements and environmental considerations were all taken on board during the design of this software, as was a broad review of the state-of-the-art in distributed storage systems.

The resulting design consisted of a hierarchy of software – the DSS layer implementing a completely distributed storage facility, and the DFS layer on top, implementing a reference file system with a centralised index. This neatly mirrors the level of abstraction on a conventional computer, where multiple file systems can be created on top of the one hard disk. For the DBE, this approach conveniently supports the concept of SMEs offering value-added file-system services as a commercial offering.

In terms of implementation, the layers of the software were subdivided into appropriate components, minimising the size of packages that needed to be deployed to the client and service side respectively. Simple-to-use helper classes were provided to minimise the coding effort required by developers integrating their applications with both the DSS and DFS. Appropriate design patterns were used within the DSS service to facilitate the swapping of individual indexing and storage components without requiring recompilation and redeployment of the entire DSS service. Before completion of the DHT, a series of indexing components were developed to facilitate development and testing of the DSS in a variety of network scenarios. Similarly, the database component of the DFS can be easily swapped-out.

Sample and graphical client applications were also developed as part of this project. As well as illustrating how the components could be used, they also provided a mechanism to test the integration of the components. Unit testing was used to verify the internal functionality of the individual software components themselves, with extensive, configurable logging and explicit exceptions implemented to facilitate debugging and troubleshooting after deployment.

The software has been successfully integrated with the DBE Execution Environment, and now provides a distributed storage facility which SME services, infrastructural DBE services and client applications can consume on this evolving research platform.

In terms of possible future work, several potential enhancements have been identified that may prove interesting and useful to pursue. Those that are tackled will largely depend on adoption, the evolution of future requirements, and performance of the system in internet deployments.

If bandwidth consumption becomes an issue, the adoption of erasure-coded blocks rather than direct partitioning of content may provide a practical solution. If the available quantity of storage becomes a concern, perhaps a

StoreConnector class could be developed that persists the content onto an alternative distributed storage system, even, potentially, BitTorrent or OceanStore.

Although encryption of content is supported by the DSS, other aspects of security have not been addressed due to larger project priorities. In future versions of the DSS it may prove appropriate to only accept content that has a certified digital identity associated with it. This could possibly be achieved by integration with the new DSS Identity management service, and would enable limitations to be put on those identities that could manipulate content, be it delete it or alter the time-to-live. Identity would also facilitate the implementation of a suitable fairness algorithm if required, to reduce or prevent the ability of users to consume more than they provide.

As additional use-cases for the DSS are conceived, the software described in this document may be adapted or extended to facilitate such new requirements. Please consult the swallow SourceForge [38] project at http://swallow.sourceforge.net for the most up-to-date status of this software.

8 Bibliography

1. DBE: See http://www.digital-ecosystem.org
2. DBE Architecture: "DBE Deliverable D21.2: Architecture Scope Document". P. Ferronato – Soluta.NET. Available at http://www.digital-ecosystem.org
3. DBE ExE: See http://swallow.sourceforge.net
4. DBE Studio: See http://dbestudio.sourceforge.net
5. DBE EvE: See http://evenet.sourceforge.net
6. Knowledge Base and Semantic Registry: "DBE Deliverable D14.5: Final P2P implementation of the DBE Knowledge Base and SR", N. Pappas, G. Kotopoulos, Y. Kotopoulos – Technical University of Crete. Available at http://www.digital-ecosystem.org
7. FADA: Federated Autonomous Directory Architecture. See http://fada.sourceforge.net/
8. DHT & Identity: "DBE Deliverable D24.3: DBE Peer-to-Peer Architecture Design", B. Biskupski, J. Sacha, J. Dowling et al – Trinity College Dublin. Available at http://www.digital-ecosystem.org
9. Accounting: "DBE Deliverable D36.2: A set of Accounting Software Building Blocks", F. Walsh – Waterford Institute of Technology. Available at http://www.digital-ecosystem.org
10. Contracts: "DBE Deliverable D32.6: DBE Model for Generic Level DBE Contracts", P. Malone, J. Finnegan – Waterford Institute of Technology. Available at http://www.digital-ecosystem.org
11. EvE: "DBE Deliverable D9.4: EvE Service Implementation", C. Masuch et al, Salzburg Technical University / London School of Economics. To be made available at http://www.digital-ecosystem.org
12. User Profiling: "DBE Deliverable D7.2, Implementation of Profiling Mechanism", C. Bartsch et al – FZI (Forschungszentrum Informatik). Available at http://www.digital-ecosystem.org
13. Indiana University Massive Data Storage System Service: "A. Shankar et al, Building and supporting a massive data infrastructure for the masses (2002)". Available in Proceedings of the 30th annual ACM SIGUCCS conference on User services, available at http://portal.acm.org/citation.cfm?id=588674
14. Original Napster: See http://en.wikipedia.org/wiki/Napster
15. FreeNet: See http://freenetproject.org/
16. BitTorrent: See http://www.bittorrent.com, http://www.bittorrent.org/protocol.html
17. PAST: See http://research.microsoft.com/~antr/PAST/default.htm
18. Gnutella: See http://www.gnutella.com/
19. Cooperative File System: F. Dabek, "A Cooperative File System", 2001. Available at http://citeseer.ist.psu.edu/dabek01cooperative.html
20. BitVault: http://research.microsoft.com/research/pubs/view.aspx?tr_id=1035
21. Google File System: See http://labs.google.com/papers/gfs.html
22. Frangipani: C. A. Thekkath et al, "Frangipani, a Scalable, Distributed File System". Available at http://www.thekkath.org/papers/frangipani.pdf

23. Petal: E. Lee, C. Thekkrath. “Petal: Distributed virtual disks”. Available in Proceedings of the 7th international Conference on Architectural Support for Programming Languages and Operating (ASPLOS 1996) p. 84-92, October 1996
24. Ivy: R. Morris, T. M. Gil, B. Chen, “Ivy: A Read/Write Peer-to-peer File System”. Available at http://pdos.csail.mit.edu/ivy/
25. OceanStore: J. Kubiatowicz et al, “OceanStore: An Architecture for Global-Scale Persistent Storage”. Available in Proceedings of the 9th international Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000), November 2000
26. Pond: S. Rhea et al, “Pond: the OceanStore Prototype”. Available in Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST '03), March 2003
27. Venti: S. Quinlan, S. Dorward, “Venti: a new Approach to Archival Storage”, available at http://plan9.bell-labs.com/sys/doc/venti.html
28. Two-Level, Self-Verifying Data: P. Eaton, H. Weatherspoon, J. Kubiatowicz, “Two-Level, Self-Verifying Data for Peer-to-Peer Storage”, 2005. Available at http://www.eecs.berkeley.edu/Pubs/TechRpts/2005/CSD-05-1401.pdf
29. Adaptive Connection Establishment: L. Xiao, Y. Liu, L. M. Ni “Improving Unstructured Peer-to-Peer Systems by Adaptive Connection Establishment”, IEEE Transactions on Computers, Vol. 54, No. 9, September 2005. Available at http://ieeexplore.ieee.org/iel5/12/31532/01471671.pdf?arnumber=1471671
30. Erasure Coding: H. Weatherspoon, J. Kubiatowicz, “Erasure Coding vs. Replication: A Quantitative Comparison”, available at http://oceanstore.org/publications/papers/pdf/erasure_iptps.pdf
31. Mojo Nation: See http://en.wikipedia.org/wiki/Mnet
32. mnet: See http://mnetproject.org/
33. Tor: See http://tor.eff.org/
34. FreeHaven: See http://freehaven.net/index.html
35. Design Patterns: Gamma et al., "Design Patterns: Elements of Reusable Object-Oriented Software", Addison Wesley, 1995
36. PostgreSql: See http://www.postgresql.org/
37. Log4j: http://logging.apache.org/log4j/docs/, accessed December 2006
38. Sourceforge: http://www.sourceforge.net, accessed December 2006

Appendix A DSS API

A.1 DSS Client

Package Summary	
org.dbe.dss	Provides client classes for accessing the DBE Distributed Storage System (DSS).

Package org.dbe.dss

Provides client classes for accessing the DBE Distributed Storage System (DSS).

See:
Description

Class Summary	
DSSConnector	Provides a proxy connected to a live DSS service.
DSSData	A helper class to simplify accessing the DBE DSS.

Exception Summary	
DSSException	Signals a DSS-specific exception has occurred.

Package org.dbe.dss Description

Provides client classes for accessing the DBE Distributed Storage System (DSS).

Interaction with the DSS is typically through the DSSData class.

Author:
Intel Ireland Ltd.
Version:
0.5

Class DSSConnector

org.dbe.dss

```
java.lang.Object
  └org.dbe.dss.DSSConnector
```

---

final public class **DSSConnector**
extends Object

Provides a proxy connected to a live DSS service.

Author:
Intel Ireland Ltd.
Version:
0.5.0

Method Summary	
static DSSService	**getDSSService**() Provides a proxy to a live DSS service.

Method Detail

getDSSService

```
public static DSSService getDSSService()
                              throws IOException
```

Provides a proxy to a live DSS service.
Returns:
connected to a live DSS Service.
Throws:
`IOException` - if an I/O error occurs

Class DSSData

org.dbe.dss

```
java.lang.Object
  └org.dbe.dss.DSSData
```

---

final public class **DSSData**
extends Object

A helper class to simplify accessing the DBE DSS.

Author:
Intel Ireland Ltd.
Version:
0.5.0

Method Summary	
static int	**getTimeToLive**(String hashcode) Retrieves the time-to-live (in seconds) of the DSS content corresponding to the hashcode supplied.
static byte[]	**read**(String hashcode) Reads the data from the DSS that corresponds to the supplied hashcode.
static byte[]	**read**(String hashcode, int offset, int len) Reads up to `len` bytes of data from the content on the DSS that corresponds to the supplied hashcode, starting at the `offset` position in that content.
static byte[]	**readEncrypted**(String algorithm, byte[] key, String hashcode) Reads the encrypted data from the DSS that corresponds to the supplied hashcode.
static byte[]	**readEncrypted**(String algorithm, byte[] key, String hashcode, int offset, int len) Reads up to `len` bytes of data from the encrypted content on the DSS that corresponds to the supplied hashcode, starting at the `offset` position in that decrypted content.
static boolean	**update**(String hashcode, int timetolive) Updates the time-to-live of the DSS content corresponding to the hashcode supplied.
static String	**write**(byte[] data, int timetolive) Writes the given data to the DSS, where it will be persisted until the supplied time-to-live expires.
static String	**writeEncrypted**(String algorithm, byte[] key, byte[] data, int timetolive) Writes the given data to the DSS, after encrypting it using the algorithm and key supplied, where it will be persisted until the supplied time-to-live expires.

Method Detail

write

```
public static String write(byte[] data,
                           int timetolive)
                    throws IOException
```

Writes the given data to the DSS, where it will be persisted until the supplied time-to-live expires.

Parameters:

`data` - the data to persist on the DSS

`timetolive` - the amount of time, in seconds, after which the data may be automatically deleted from the DSS

Returns:

a hashcode that can be used to retrieve the data in the future.

Throws:

`IOException` - if an I/O error occurs

---

read

```
public static byte[] read(String hashcode)
                  throws IOException
```

Reads the data from the DSS that corresponds to the supplied hashcode.

Parameters:

`hashcode` - the hashcode of the data to be retrieved

Returns:

a byte array containing the requested data.

Throws:

`IOException` - if an I/O error occurs

---

read

```
public static byte[] read(String hashcode,
                          int offset,
                          int len)
                  throws IOException
```

Reads up to `len` bytes of data from the content on the DSS that corresponds to the supplied hashcode, starting at the `offset` position in that content. If there are not enough such bytes in the content then less than `len` bytes will be returned.

Parameters:

`hashcode` - the hashcode of the data to be retrieved

`offset` - the start offset of the data

`len` - the maximum number of bytes to be read

Returns:

a byte array containing the requested data.

Throws:

`IOException` - if an I/O error occurs

---

getTimeToLive

```
public static int getTimeToLive(String hashcode)
                         throws IOException
```

Retrieves the time-to-live (in seconds) of the DSS content corresponding to the hashcode supplied.

Parameters:

`hashcode` - the hashcode of the data whose time-to-live is to be retrieved

Returns:

the amount of time, in seconds, after which the data may be automatically deleted from the DSS.

Throws:

`IOException` - if an I/O error occurs

---

update

```
public static boolean update(String hashcode,
                             int timetolive)
                      throws IOException
```

Updates the time-to-live of the DSS content corresponding to the hashcode supplied.

Parameters:

`hashcode` - the hashcode of the data whose time-to-live is to be updated

`timetolive` - the amount of time, in seconds, after which the data may be automatically deleted from the DSS

Returns:

true if the time-to-live was updated successfully.

Throws:

`IOException` - if an I/O error occurs

---

readEncrypted

```
public static byte[] readEncrypted(String algorithm,
                                   byte[] key,
                                   String hashcode)
                            throws IOException,
                                   InvalidKeyException,
                                   NoSuchAlgorithmException,
                                   NoSuchPaddingException,
                                   IllegalBlockSizeException,
                                   BadPaddingException
```

Reads the encrypted data from the DSS that corresponds to the supplied hashcode. The content is decrypted using the algorithm and key supplied.

Parameters:

`algorithm` - the algorithm to use to decrypt the data

`key` - the key to decrypt the data

`hashcode` - the hashcode of the data to be retrieved

Returns:

a byte array containing the requested data.

Throws:

`IOException` - if an I/O error occurs

`InvalidKeyException` - if an invalid encryption key error occurs

`NoSuchAlgorithmException` - if an invalid encryption algorithm error occurs

`NoSuchPaddingException` - if an missing encryption padding error occurs

`IllegalBlockSizeException` - if an illegal encryption block size error occurs

`BadPaddingException` - if an encryption padding error occurs

---

readEncrypted

```
public static byte[] readEncrypted(String algorithm,
                                   byte[] key,
                                   String hashcode,
                                   int offset,
                                   int len)
                            throws IOException,
                                   InvalidKeyException,
                                   NoSuchAlgorithmException,
                                   NoSuchPaddingException,
                                   IllegalBlockSizeException,
                                   BadPaddingException
```

Reads up to `len` bytes of data from the encrypted content on the DSS that corresponds to the supplied hashcode, starting at the `offset` position in that decrypted content. The content is decrypted using the

algorithm and key supplied. If there are not enough such bytes in the content then less than `len` bytes will be returned.

Parameters:

`algorithm` - the algorithm to use to decrypt the data
`key` - the key to decrypt the data
`hashcode` - the hashcode of the data to be retrieved
`offset` - the start offset of the data
`len` - the maximum number of bytes to be read

Returns:

a byte array containing the requested data.

Throws:

`IOException` - if an I/O error occurs
`InvalidKeyException` - if an invalid encryption key error occurs
`NoSuchAlgorithmException` - if an invalid encryption algorithm error occurs
`NoSuchPaddingException` - if an missing encryption padding error occurs
`IllegalBlockSizeException` - if an illegal encryption block size error occurs
`BadPaddingException` - if an encryption padding error occurs

---

writeEncrypted

```
public static String writeEncrypted(String algorithm,
                                    byte[] key,
                                    byte[] data,
                                    int timetolive)
                             throws IOException,
                                    InvalidKeyException,
                                    NoSuchAlgorithmException,
                                    NoSuchPaddingException,
                                    IllegalBlockSizeException,
                                    BadPaddingException
```

Writes the given data to the DSS, after encrypting it using the algorithm and key supplied, where it will be persisted until the supplied time-to-live expires.

Parameters:

`algorithm` - the algorithm to use to encrypt the data
`key` - the key to encrypt the data
`data` - the data to persist on the DSS
`timetolive` - the amount of time, in seconds, after which the data may be automatically deleted from the DSS

Returns:

a hashcode that can be used to retrieve the data in the future.

Throws:

`IOException` - if an I/O error occurs
`InvalidKeyException` - if an invalid encryption key error occurs
`NoSuchAlgorithmException` - if an invalid encryption algorithm error occurs
`NoSuchPaddingException` - if an missing encryption padding error occurs
`IllegalBlockSizeException` - if an illegal encryption block size error occurs
`BadPaddingException` - if an encryption padding error occurs

Class DSSException

org.dbe.dss

```
java.lang.Object
  └java.lang.Throwable
      └java.lang.Exception
          └java.io.IOException
              └org.dbe.dss.DSSException
```

All Implemented Interfaces:

Serializable

---

public class **DSSException**
extends IOException

Signals a DSS-specific exception has occurred.

Author:

Intel Ireland Ltd.

Version:

0.5.0

Constructor Summary
DSSException()   Creates a new exception.
DSSException(String message)   Creates a new exception with the given message.

Constructor Detail

DSSException

```
public DSSException()
```

Creates a new exception.

DSSException

```
public DSSException(String message)
```

Creates a new exception with the given message.

A.2 DSS Service

Package org.dbe.dss.service

Interface Summary	
DSSService	Describes an interface for a service which implements a DBE Distributed File System (DSS).

Interface DSSService

org.dbe.dss.service

---

public interface **DSSService**

Describes an interface for a service which implements a DBE Distributed File System (DSS).

Author:
Intel Ireland Ltd.
Version:
0.5.0

Method Summary	
String	**getStoreID**() Retrieves the unique identifier of the local store to which this DSS service is persisting content.
long	**getTimeToLive**(String hashcode) Retrieves the time-to-live for the content assoociated with the identifier supplied.
boolean	**isAlive**() Confirms that the service is responding to invocations.
byte[]	**read**(String hashcode) Reads the content associated with the identifier supplied from the DSS.
byte[]	**read**(String hashcode, int start, int len) Reads the required subset of the content associated with the identifier supplied from the DSS.
boolean	**update**(String hashcode, long timeToLive) Updates the time-to-live for the content assoociated with the identifier supplied.
String	**write**(byte[] content, long timetolive) Writes content to the DSS, assigning it the supplied time-to-live.
String	**write**(byte[] content, long timeToLive, boolean replicate) Writes content to the DSS, assigning it the supplied time-to-live.

Method Detail

write

```
public String write(byte[] content,
                    long timetolive)
             throws DSSException
```

Writes content to the DSS, assigning it the supplied time-to-live. The content is replicate before the method returns.

Parameters:

`content` - the data to be persisted

Returns:

the unique identifier for the content.

Throws:

`DSSException` - if an error occurs during invocation

---

write

```
public String write(byte[] content,
                    long timeToLive,
                    boolean replicate)
             throws DSSException
```

Writes content to the DSS, assigning it the supplied time-to-live. If requested, the method returns before replication has completed.

Parameters:

`content` - the data to be persisted

`timeToLive` - the amount of time, in seconds, after which the content may be automatically deleted from the DSS. It must be greater than 0.

`replicate` - whether the method should wait for replication to complete before returning

Returns:

the unique identifier for the content.

Throws:

`DSSException` - if an error occurs during invocation

---

read

```
public byte[] read(String hashcode)
            throws DSSException
```

Reads the content associated with the identifier supplied from the DSS.

Parameters:

`hashcode` - the unique identifier for the content

Returns:

the bytes of data read.

Throws:

`DSSException` - if an error occurs during invocation

---

read

```
public byte[] read(String hashcode,
                   int start,
                   int len)
            throws DSSException
```

Reads the required subset of the content associated with the identifier supplied from the DSS. The subset is specified in terms of an offset measured from the start of the content, and the length of the subset.

Parameters:

`hashcode` - the unique identifier for the content

`start` - the start offset in the content

`len` - the number of bytes to read

Returns:

the byte of data read.

Throws:

`DSSException` - if an error occurs during invocation

---

update

```
public boolean update(String hashcode,
                      long timeToLive)
               throws DSSException
```

Updates the time-to-live for the content assoociated with the identifier supplied.

Parameters:

`hashcode` - the unique identifier for the content

`timeToLive` - the amount of time, in seconds, after which the content may be automatically deleted from the DSS. It must be greater than 0.

Returns:

true if the update was successful, otherwise false.

Throws:

`DSSException` - if an error occurs during invocation

---

getTimeToLive

```
public long getTimeToLive(String hashcode)
                   throws DSSException
```

Retrieves the time-to-live for the content assoociated with the identifier supplied.

Parameters:

`hashcode` - the unique identifier for the content

Returns:

the amount of time, in seconds, after which the content will be automatically deleted.

Throws:

`DSSException` - if an error occurs during invocation

---

isAlive

```
public boolean isAlive()
```

Confirms that the service is responding to invocations.
Returns:
true.

---

getStoreID

```
public String getStoreID()
```

Retrieves the unique identifier of the local store to which this DSS service is persisting content.
Returns:
the unique store-identifier assigned to the local store.

Appendix B DFS API

B.2 DFS Client

Package Summary	
org.dbe.dfs	Provides client classes for accessing a DBE File System (DFS).

Package org.dbe.dfs

Provides client classes for accessing a DBE File System (DFS).

See:
Description

Class Summary	
DFSConnector	
DFSFile	An abstract representation of file and directory pathnames in the DFS.
DFSInputStream	An InputStream used to read data from a `DFSFile` , a file in the DFS.
DFSOutputStream	An OutputStream used to write data to a `DFSFile` , a file in the DFS.
DFSDirectoryInfo	Contains information about the total number of files and the total amount of data contained in a directory, including all its subdirectories.
DFSFileInfo	Contains information about a file in the DFS.

Exception Summary	
DFSException	Implements an explicit DFS exception.

Package org.dbe.dfs Description

Provides client classes for accessing a DBE File System (DFS).

Interactions with the DFS are primarily through the DFSFile, DFSOutputStream and DFSInputStream classes. These work in a similar way to the standard `java.io` classes.

Author:
Intel Ireland Ltd.
Version:
0.4

Class DFSConnector

org.dbe.dfs

```
java.lang.Object
  └org.dbe.dfs.DFSConnector
```

---

public class **DFSConnector**
extends Object

Author:
Intel Ireland Ltd.
Version:
0.4.0 Returns an instance of a DBE service

Field Summary	
private static org.dbe.dfs.service.DFSService	**dfs**

Constructor Summary
DFSConnector()

Method Summary	
static org.dbe.dfs.service.DFSService	**getDFSService**() Returns an instance of DFSService

Field Detail

dfs

```
private static org.dbe.dfs.service.DFSService dfs
```

Constructor Detail

DFSConnector

```
public DFSConnector()
```

Method Detail

getDFSService

```
public static org.dbe.dfs.service.DFSService getDFSService()
                                                  throws DFSException
```

Returns an instance of DFSService
Returns:
DFSService

Throws:
DFSException - if DFS service could not be found

Class DFSException

org.dbe.dfs

```
java.lang.Object
  └java.lang.Throwable
      └java.lang.Exception
          └java.io.IOException
              └org.dbe.dfs.DFSException
```

All Implemented Interfaces:
Serializable

---

public class **DFSException**
extends IOException

Implements an explicit DFS exception.

Author:
Intel Ireland Ltd.
Version:
0.4.0

Constructor Summary
DFSException() Creates a new exception.
DFSException(String message) Creates a new exception with the given message.

Constructor Detail

DFSException

```
public DFSException()
```

Creates a new exception.

DFSException

```
public DFSException(String message)
```

Creates a new exception with the given message.

Class DFSFile

org.dbe.dfs

```
java.lang.Object
  └org.dbe.dfs.DFSFile
```

---

public class **DFSFile**
extends Object

An abstract representation of file and directory pathnames in the DFS.

DFSFile is equivalent to the class `java.io.File`, with several important differences.

An abstract DFS pathname has two components:

1. An optional prefix which is a DFS path separator ('\', '\\', '/', or '//')
2. A sequence of zero or more string names.

Each name in the abstract pathname is a directory, except for the last one which may be a directory or a file. These names are separated by a path separator. The following restrictions apply to these names:

- A name can contain any printable Unicode characters except for the following which are forbidden: " * / : ; < > ? \ |
- A name cannot be more than 250 characters in length
- A name cannot contain only a sequence of the period ('.') character, and cannot end with this character
- A name cannot contain only a sequence of the space character, and cannot end with this character

There is no concept of 'current directory' in the DFS and so absolute (complete) pathnames must always be supplied to DFSFile methods. An absolute pathname can begin with a prefix (path separator).

Although DFSFile methods accept '\', '\\', '/', and '//' as valid path separators, any pathnames returned by DFS methods will use the path separator in the pathSeparator field. Pathnames returned will also begin with this prefix.

Directory and file names in the DFS are not case-sensitive, however case is preserved. "foo.txt" is the same file as "FoO.txt".

Every file in the DFS has a time-to-live value associated with it. The file will only exist in the DFS for the time-to-live amount of time. Time-to-live is expressed in seconds. The default time-to-live is defined by the DEFAULT_TIME_TO_LIVE field.

Author:
Intel Ireland Ltd.
Version:
0.4.0

Field Summary	
static final long	**DEFAULT_TIME_TO_LIVE** Default time-to-live (90 days) in seconds.
private String	**dfsFilePath** DFS path of the file.

static final long	**INFINITE_TIME_TO_LIVE** Infinite time-to-live.
static final String	**pathSeparator** The system-dependent path-separator character, represented as a string for convenience.
static final char	**pathSeparatorChar** The system-dependent path-separator character.
private static final String	**REGEX_VALID_SEPARATORS** A regular expression defining valid path separators.
static final String	**separator** The system-dependent default name-separator character, represented as a string for convenience.
static final char	**separatorChar** The system-dependent default name-separator character.

Constructor Summary

DFSFile(String dfsPath) Creates a new DFSFile instance by converting the given pathname string into an abstract pathname.

Method Summary

final boolean	**canRead**() Tests whether the application can read the file denoted by this abstract pathname.
final boolean	**canWrite**() Tests whether the application can modify the contents of the file denoted by this abstract pathname.
final boolean	**createNewFile**() Atomically creates a new, empty file with this abstract pathname and a default time-to-live.
final boolean	**createNewFile**(long timeToLive) Atomically creates a new, empty file with this abstract pathname and the time-to-live specified.
final boolean	**delete**() Deletes the file or directory denoted by this abstract pathname.
final org.dbe.dfs.DFSDirectoryInfo	**directoryInfo**() Returns information about the directory denoted by the abstract pathname supplied.
final boolean	**equals**(Object obj) Tests this abstract pathname for equality with the given object.
final boolean	**exists**() Tests whether the file or directory denoted by this abstract pathname exists.
final DFSInputStream	**getDFSInputStream**() Returns a DFSInputStream connected to this abstract pathname, allowing reading from the file.
final DFSOutputStream	**getDFSOutputStream**() Returns a DFSOutputStream connected to this abstract pathname whose existing contents, if any, will be replaced.

final DFSOutputStream	**getDFSOutputStream**(boolean append) Returns a DFSOutputStream connected to this abstract pathname whose existing contents, if any, can optionally be appended to.
private org.dbe.dfs.service.DFSService	**getDFSService**() Returns an instance of the DFS service and delays the request if necessary.
final String	**getName**() Returns the name of the file or directory denoted by this abstract pathname.
final String	**getPath**() Returns the pathname of this abstract pathname.
private String	**getRegex**(String str) Returns a regular expression that matches the given string.
final long	**getTimeToLive**() Retrieves the time-to-live value in seconds.
final int	**hashCode**() Computes a hash code for this abstract pathname.
final boolean	**isDirectory**() Tests whether the file denoted by this abstract pathname is a directory.
final boolean	**isFile**() Tests whether the file denoted by this abstract pathname is a normal file.
final long	**length**() Returns the length of the file denoted by this abstract pathname.
final String[]	**list**() Returns an array of strings naming the files and directories in the directory denoted by this abstract pathname.
final DFSFile[]	**listFiles**() Returns an array of abstract pathnames denoting the files in the directory denoted by this abstract pathname.
final org.dbe.dfs.DFSFileInfo[]	**listFilesInfo**() Returns an array of information about the files and directories in the directory denoted by the abstract pathname supplied.
private String	**localiseSeparator**(String path) Changes path separators from any DFS valid separator to the local separator in this machine, given by File.separator
final boolean	**mkdir**() Creates the directory named by this abstract pathname.
final boolean	**mkdirs**() Creates the directory named by this abstract pathname, including any necessary but nonexistent parent directories.
private String	**normaliseCase**(String path) Normalises case in DFS path names in order to be able to compare them.
private String	**normaliseSeparator**(String dfsPath) Replaces the allowed separators by the local separator character (i.e.
final boolean	**renameTo**(DFSFile dest) Renames the file denoted by this abstract pathname.
final void	**setTimeToLive**(long timeToLive) Sets the time-to-live value in seconds.

final String	**toString**() Returns the pathname of this abstract pathname.

Field Detail

pathSeparator

```
public static final String pathSeparator
```

The system-dependent path-separator character, represented as a string for convenience. This string contains a single character, namely pathSeparatorChar.

---

pathSeparatorChar

```
public static final char pathSeparatorChar
```

The system-dependent path-separator character. This character is used to separate filenames in a sequence of files given as a path list. This character matches the conventional separator character as returned by `java.io.File.pathSeparatorChar`.

---

separator

```
public static final String separator
```

The system-dependent default name-separator character, represented as a string for convenience. This string contains a single character, namely separatorChar.

---

separatorChar

```
public static final char separatorChar
```

The system-dependent default name-separator character. This character matches the conventional separator character as returned by `link java.io.File.separatorChar`.

DEFAULT_TIME_TO_LIVE

```
public static final long DEFAULT_TIME_TO_LIVE
```

Default time-to-live (90 days) in seconds.

INFINITE_TIME_TO_LIVE

```
public static final long INFINITE_TIME_TO_LIVE
```

Infinite time-to-live.

---

dfsFilePath

```
private String dfsFilePath
```

DFS path of the file.

REGEX_VALID_SEPARATORS

```
private static final String REGEX_VALID_SEPARATORS
```

A regular expression defining valid path separators. The actual path separators of \\, \, // and / are escaped in this String.

Constructor Detail

DFSFile

```
public DFSFile(String dfsPath)
```

Creates a new DFSFile instance by converting the given pathname string into an abstract pathname. If the given string is the empty string, then the result is the empty abstract pathname.

Method Detail

createNewFile

```
public final boolean createNewFile()
                            throws IOException,
                                   DFSException
```

Atomically creates a new, empty file with this abstract pathname and a default time-to-live. The file is created if and only if a file with this name does not yet exist.

Returns:

`true` if the named file does not exist and was successfully created; `false` if the named file already exists.

Throws:

`IOException` - if an I/O error occurs

`DFSException` - if a DFS specific error occurs

---

createNewFile

```
public final boolean createNewFile(long timeToLive)
                            throws IOException,
                                   DFSException
```

Atomically creates a new, empty file with this abstract pathname and the time-to-live specified. The file is created if and only if a file with this name does not yet exist.

Parameters:

`timeToLive` - time-to-live for the new file.

Returns:

`true` if the named file does not exist and was successfully created; `false` if the named file already exists.

Throws:

`IOException` - if an I/O error occurs

`DFSException` - if a DFS specific error occurs

---

canRead

```
public final boolean canRead()
                     throws DFSException
```

Tests whether the application can read the file denoted by this abstract pathname.

Returns:

`true` if and only if the file specified by this abstract pathname exists and can be read by the application; `false` otherwise.

Throws:

`DFSException` - if a DFS specific error occurs

---

canWrite

```
public final boolean canWrite()
                      throws DFSException
```

Tests whether the application can modify the contents of the file denoted by this abstract pathname.

Returns:

`true` if and only if the file system actually contains a file denoted by this abstract pathname and the application is allowed to write to the file; `false` otherwise.

Throws:

`DFSException` - if a DFS specific error occurs

---

exists

```
public final boolean exists()
                     throws DFSException
```

Tests whether the file or directory denoted by this abstract pathname exists.

Returns:

`true` if and only if the file or directory denoted by this abstract pathname exists; `false` otherwise.

Throws:

`DFSException` - if a DFS specific error occurs

---

delete

```
public final boolean delete()
                    throws DFSException
```

Deletes the file or directory denoted by this abstract pathname. If this pathname denotes a directory, then the directory must be empty in order to be deleted.

Returns:

`true` if and only if the file or directory is successfully deleted; `false` otherwise.

Throws:

`DFSException` - if a DFS specific error occurs

---

setTimeToLive

```
public final void setTimeToLive(long timeToLive)
                         throws IllegalArgumentException,
                                IOException,
                                DFSException
```

Sets the time-to-live value in seconds.

Parameters:

`timeToLive` - the number of seconds for which the file will live.

Throws:

`IllegalArgumentException` - if timeToLive is negative

`IOException` - if the time-to-live can not be set for this file, the file name is incorrect or the file does not exist

`DFSException` - if a DFS specific error occurs

---

getTimeToLive

```
public final long getTimeToLive()
                         throws DFSException,
                                FileNotFoundException,
                                IOException
```

Retrieves the time-to-live value in seconds.

Returns:

the number of seconds for which the file will live, or INFINITE_TIME_TO_LIVE if the file has an infinite time-to-live. All directories have an infinite time-to-live.

Throws:

`DFSException` - if a DFS specific error occurs

`FileNotFoundException` - if the abstract pathname does not exist

`IOException` - if an IO error occurs

---

length

```
public final long length()
                  throws DFSException
```

Returns the length of the file denoted by this abstract pathname. The return value is unspecified if this pathname denotes a directory.

Returns:
the length, in bytes, of the file denoted by this abstract pathname, or 0L if the file does not exist.
Throws:
DFSException - if a DFS specific error occurs

---

renameTo

```
public final boolean renameTo(DFSFile dest)
                       throws DFSException
```

Renames the file denoted by this abstract pathname.
Parameters:
dest - the new abstract pathname for the named file.
Returns:
true if and only if the renaming succeeded; false otherwise.
Throws:
DFSException - if a DFS specific error occurs

---

getName

```
public final String getName()
```

Returns the name of the file or directory denoted by this abstract pathname. This is just the last name in the pathname's name sequence. If the pathname's name sequence is empty, then the empty string is returned.
Returns:
the name of the file or directory denoted by this abstract pathname, or the empty string if this pathname's name sequence is empty.

---

getPath

```
public final String getPath()
```

Returns the pathname of this abstract pathname.
Returns:
the string form of the abstract pathname.

---

mkdir

```
public final boolean mkdir()
                    throws DFSException
```

Creates the directory named by this abstract pathname.
Returns:
true if and only if the directory was created; false otherwise.
Throws:
DFSException - if a DFS specific error occurs

mkdirs

```
public final boolean mkdirs()
                    throws DFSException
```

Creates the directory named by this abstract pathname, including any necessary but nonexistent parent directories. Note that if this operation fails it may have succeeded in creating some of the necessary parent directories.

Returns:

`true` if and only if the directory was created, along with all necessary parent directories; `false` otherwise.

Throws:

`DFSException` - if a DFS specific error occurs

---

isFile

```
public final boolean isFile()
                    throws DFSException
```

Tests whether the file denoted by this abstract pathname is a normal file. A file is normal if it is not a directory and, in addition, satisfies other system-dependent criteria. Any non-directory file created by a Java application is guaranteed to be a normal file.

Returns:

`true` if and only if the file denoted by this abstract pathname exists and is a normal file; `false` otherwise.

Throws:

`DFSException` - if a DFS specific error occurs

---

isDirectory

```
public final boolean isDirectory()
                         throws DFSException
```

Tests whether the file denoted by this abstract pathname is a directory.

Returns:

`true` if and only if the file denoted by this abstract pathname exists and is a directory; `false` otherwise.

Throws:

`DFSException` - if a DFS specific error occurs

---

list

```
public final String[] list()
                   throws DFSException
```

Returns an array of strings naming the files and directories in the directory denoted by this abstract pathname. If this abstract pathname does not denote a directory, then this method returns `null`. Otherwise an array of strings is returned, one for each file or directory in the directory. Names denoting the directory

itself and the directory's parent directory are not included in the result. Each string is a file name rather than a complete path.

Returns:

an array of strings naming the files and directories in the directory denoted by this abstract pathname. The array will be empty if the directory is empty. Returns `null` if this abstract pathname does not denote a directory, or if an I/O error occurs.

Throws:

`DFSException` - if a DFS specific error occurs

---

listFiles

```
public final DFSFile[] listFiles()
                        throws DFSException
```

Returns an array of abstract pathnames denoting the files in the directory denoted by this abstract pathname. If this abstract pathname does not denote a directory, then this method returns `null`. Otherwise an array of DFSFile objects is returned, one for each file or directory in the directory. Pathnames denoting the directory itself and the directory's parent directory are not included in the result.

Returns:

an array of abstract pathnames denoting the files and directories in the directory denoted by this abstract pathname. The array will be empty if the directory is empty. Returns `null` if this abstract pathname does not denote a directory, or if an I/O error occurs.

Throws:

`DFSException` - if a DFS specific error occurs

---

listFilesInfo

```
public final org.dbe.dfs.DFSFileInfo[] listFilesInfo()
                                            throws DFSException
```

Returns an array of information about the files and directories in the directory denoted by the abstract pathname supplied. If the abstract pathname supplied does not denote a directory, then this method returns `null`. Otherwise an array of DFSFileInfo is returned, one for each file or directory in the directory.

Returns:

an array with information about all the files, or `null` if this abstract pathname does not denote a directory.

Throws:

`DFSException` - if a DFS specific error occurs

---

getDFSInputStream

```
public final DFSInputStream getDFSInputStream()
                                     throws FileNotFoundException,
                                            DFSException
```

Returns a DFSInputStream connected to this abstract pathname, allowing reading from the file.

Returns:

a DFSInputStream connected to this abstract pathname.

Throws:

`FileNotFoundException` - if an error occurs

DFSException - if a DFS specific error occurs

---

getDFSOutputStream

```
public final DFSOutputStream getDFSOutputStream()
                                        throws FileNotFoundException,
                                               DFSException
```

Returns a DFSOutputStream connected to this abstract pathname whose existing contents, if any, will be replaced.

Returns:

a DFSOutputStream connected to this abstract pathname.

Throws:

FileNotFoundException - if the abstract pathname does not exist

DFSException - if a DFS specific error occurs

---

getDFSOutputStream

```
public final DFSOutputStream getDFSOutputStream(boolean append)
                                        throws FileNotFoundException,
                                               DFSException
```

Returns a DFSOutputStream connected to this abstract pathname whose existing contents, if any, can optionally be appended to.

Parameters:

append - if true, then bytes will be written to the end of the file rather than the beginning.

Returns:

a DFSOutputStream connected to this abstract pathname.

Throws:

FileNotFoundException - if the abstract pathname does not exist

DFSException - if a DFS specific error occurs

---

directoryInfo

```
public final org.dbe.dfs.DFSDirectoryInfo directoryInfo()
                                              throws DFSException,
                                                     IOException
```

Returns information about the directory denoted by the abstract pathname supplied. If the abstract pathname supplied does not denote a directory, then this method returns null. Otherwise a DFSDirectoryInfo is returned.

Returns:

information on the directory denoted by this abstract pathname, or null if this abstract pathname does not denote a directory.

Throws:

DFSException - if a DFS specific error occurs

IOException - if path is invalid or an IO error occurs while performing the operation

---

hashCode

```
public final int hashCode()
```

Computes a hash code for this abstract pathname. The hash code is equal to the exclusive-or of its pathname string, converted to lower case, and the decimal value 1234321.

Overrides:

`hashCode` in class `Object`

Returns:

a hash code for this abstract pathname.

---

equals

```
public final boolean equals(Object obj)
```

Tests this abstract pathname for equality with the given object. Returns `true` if and only if the argument is not `null` and is an abstract pathname that denotes the same file or directory as this abstract pathname. The DFS is not case sensitive.

Overrides:

`equals` in class `Object`

Parameters:

`obj` - the object to be compared with this abstract pathname.

Returns:

`true` if and only if the objects are the same; `false` otherwise.

---

toString

```
public final String toString()
```

Returns the pathname of this abstract pathname. This is just the string returned by the getPath() method.

Overrides:

`toString` in class `Object`

Returns:

the string form of this abstract pathname.

---

normaliseCase

```
private String normaliseCase(String path)
```

Normalises case in DFS path names in order to be able to compare them.

Parameters:

`path` - DFS path name

Returns:

the normalised DFS path (beginning with a separator and in lower case

---

normaliseSeparator

```
private String normaliseSeparator(String dfsPath)
```

Replaces the allowed separators by the local separator character (i.e. File.separator). Removes a trailing separator and prepends a separator if necessary.

Parameters:

`dfsPath` - the DFS path

Returns:

the path with all the separators replaced by the normal separator

See Also:

`File.separator`

---

localiseSeparator

```
private String localiseSeparator(String path)
```

Changes path separators from any DFS valid separator to the local separator in this machine, given by File.separator

Parameters:

`path` - the DFS path name

Returns:

the DFS path name with local file system separators

See Also:

`File.separator`

---

getRegex

```
private String getRegex(String str)
```

Returns a regular expression that matches the given string. It is done by escaping all the occurrences of the "\" character.

Parameters:

`str` - the string

Returns:

the regular expression with "\" escaped

---

getDFSService

```
private org.dbe.dfs.service.DFSService getDFSService()
                                              throws DFSException
```

Returns an instance of the DFS service and delays the request if necessary.

Returns:

the DFS service.

Throws:

`DFSException` - if the instance can not be obtained

Class DFSInputStream

org.dbe.dfs

```
java.lang.Object
  └java.io.InputStream
      └org.dbe.dfs.DFSInputStream
```

---

public class **DFSInputStream**
extends InputStream

An InputStream used to read data from a `DFSFile` , a file in the DFS.

Author:
Intel Ireland Ltd.
Version:
0.4.0

Field Summary	
private byte[]	**buffer** Buffer of bytes read since the last mark was set.
private long	**bufferPosition** In reset mode, reading position in the buffer (or `-1` if not in reset mode).
private boolean	**closed** True if the stream was closed.
private long	**currentPosition** Current position in the input stream (position of the next byte to be read).
private String	**dfsFilePath** DFS path of the file accessed by this stream.
private org.dbe.dfs.service.DFSService	**dfsService** Reference to the DFSService implementation to be used.
private long	**markPosition** Current mark (`-1` if no mark was set).

Constructor Summary	
public	**DFSInputStream**(String name) Creates a new DFSInputStream pointing to the supplied DFS path.
protected	**DFSInputStream**(String name, org.dbe.dfs.service.DFSService service) Creates a new DFSInputStream pointing to the supplied DFS path, using the given DFS Service.
public	**DFSInputStream**(DFSFile file) Creates a new DFSInputStream pointing to the supplied `DFSFile`.

Method Summary	
private void	**advancePosition**(long n) Advances the current position.
final int	**available**() Returns the number of bytes that can be read from this file input stream without blocking.
final void	**close**() Closes this input stream.
final void	**mark**(int readlimit) Marks the current position in this input stream.
final boolean	**markSupported**() Marks are supported in this implementation.
final int	**read**() Reads a byte of data from this input stream.
final int	**read**(byte[] b) Reads up to `b.length` bytes of data from this input stream into an array of bytes.
final int	**read**(byte[] b, int arrayOff, int len) Reads up to `len` bytes of data from this input stream into an array of bytes.
private int	**readInternal**(byte[] b, int arrayOff, int len) Internal method for reading directly from file, without using marks or buffer
private void	**removeMark**() Removes the mark in the buffer being read.
final void	**reset**() Repositions this stream to the position at the time the `mark` method was last called on this input stream.
final long	**seek**(long pos) Sets the stream offset, measured from the beginning of the file, at which the next read occurs.
final long	**skip**(long n) Skips over and discards `n` bytes of data from the input stream.

Field Detail

currentPosition

```
private long currentPosition
```

Current position in the input stream (position of the next byte to be read).

---

markPosition

```
private long markPosition
```

Current mark (`-1` if no mark was set).

---

buffer

```
private byte[] buffer
```

Buffer of bytes read since the last mark was set.

---

bufferPosition

```
private long bufferPosition
```

In reset mode, reading position in the buffer (or `-1` if not in reset mode).

---

dfsFilePath

```
private String dfsFilePath
```

DFS path of the file accessed by this stream.

---

dfsService

```
private org.dbe.dfs.service.DFSService dfsService
```

Reference to the DFSService implementation to be used.

---

closed

```
private boolean closed
```

True if the stream was closed. If `true` all the methods invoked on this object will fail.

Constructor Detail

DFSInputStream

```
protected DFSInputStream(String name,
                         org.dbe.dfs.service.DFSService service)
                  throws FileNotFoundException,
                         DFSException
```

Creates a new DFSInputStream pointing to the supplied DFS path, using the given DFS Service.

Throws:

`FileNotFoundException`
`DFSException`

DFSInputStream

```
public DFSInputStream(String name)
               throws FileNotFoundException,
                      DFSException
```

Creates a new DFSInputStream pointing to the supplied DFS path.
Throws:
`FileNotFoundException`
`DFSException`

DFSInputStream

```
public DFSInputStream(DFSFile file)
               throws FileNotFoundException,
                      DFSException
```

Creates a new DFSInputStream pointing to the supplied `DFSFile`.
Throws:
`FileNotFoundException`
`DFSException`

Method Detail

removeMark

```
private void removeMark()
```

Removes the mark in the buffer being read.

---

advancePosition

```
private void advancePosition(long n)
```

Advances the current position. It changes attributes in a consistent manner with the current mark mode.
Parameters:
`n` - number of bytes to advance

---

seek

```
public final long seek(long pos)
                throws IOException
```

Sets the stream offset, measured from the beginning of the file, at which the next read occurs. If the stream cannot be moved to the requested position, the actual position moved-to is returned.
Parameters:
`pos` - the position within the stream from which the next read should occur
Returns:
the position the stream is actually set to, measured from the beginning of the file.

Throws:
`IOException` - if an I/O error occurs

---

read

```
public final int read()
                 throws IOException,
                        DFSException
```

Reads a byte of data from this input stream.
Overrides:
`read` in class `InputStream`
Returns:
the next byte of data, or `-1` if the end of the file is reached.
Throws:
`IOException` - if an I/O error occurs
`DFSException` - if a DFS specific error occurrs
See Also:
`InputStream.read()`

---

available

```
public final int available()
                    throws IOException,
                           DFSException
```

Returns the number of bytes that can be read from this file input stream without blocking.
Overrides:
`available` in class `InputStream`
Returns:
the number of bytes that can be read from this file input stream without blocking.
Throws:
`IOException` - if an I/O error occurs
`DFSException` - if a DFS specific error occurs
See Also:
`InputStream.available()`

---

close

```
public final void close()
                  throws IOException
```

Closes this input stream. Any subsequent invocations of any other method of this object will raise an `IOException`.
Overrides:
`close` in class `InputStream`
Throws:
`IOException` - if an I/O error occurs
See Also:
`InputStream.close()`

mark

```
public final void mark(int readlimit)
```

Marks the current position in this input stream. A subsequent call to the `reset` method repositions this stream at the last marked position so that subsequent reads re-read the same bytes.

The `readlimit` argument tells this input stream to allow that many bytes to be read before the mark position gets invalidated.

The general contract of `mark` is that, if the method `markSupported` returns `true`, the stream somehow remembers all the bytes read after the call to `mark` and stands ready to supply those same bytes again if and whenever the method `reset` is called. However, the stream is not required to remember any data at all if more than `readlimit` bytes are read from the stream before `reset` is called.

Overrides:

`mark` in class `InputStream`

Parameters:

`readlimit` - the maximum limit of bytes that can be read before the mark position becomes invalid

See Also:

`reset()`, `InputStream.reset()`, `InputStream.mark(int)`

---

markSupported

```
public final boolean markSupported()
```

Marks are supported in this implementation. Returns always `true`.

Overrides:

`markSupported` in class `InputStream`

Returns:

`true`

See Also:

`InputStream.markSupported()`

---

read

```
public final int read(byte[] b,
                      int arrayOff,
                      int len)
               throws IOException,
                      DFSException,
                      IllegalArgumentException,
                      ArrayIndexOutOfBoundsException
```

Reads up to `len` bytes of data from this input stream into an array of bytes.

Overrides:

`read` in class `InputStream`

Parameters:

`b` - the buffer into which the data is read

`arrayOff` - the start offset of the data

`len` - the maximum number of bytes read

Returns:
the total number of bytes read into the buffer, or `-1` if there is no more data because the end of the file has been reached.
Throws:
`IOException` - if an I/O error occurs
`DFSException` - if a DFS specific error occurs
`IllegalArgumentException` - if arrayOff or len are negative
`ArrayIndexOutOfBoundsException` - if the limits of the array b are exceeded
See Also:
`InputStream.read(byte[], int, int)`

---

readInternal

```
private int readInternal(byte[] b,
                         int arrayOff,
                         int len)
                  throws IOException,
                         DFSException
```

Internal method for reading directly from file, without using marks or buffer
Parameters:
`b` - the array of bytes
`arrayOff` - the offset in this array
`len` - the length of bytes to be read
Returns:
the number of bytes read (or `-1` if EOF)
Throws:
`IOException` - if an I/O error occurs
`DFSException` - if a DFS specific error occurs
See Also:
`InputStream.read(byte[], int, int)`

---

read

```
public final int read(byte[] b)
               throws IOException,
                      DFSException
```

Reads up to `b.length` bytes of data from this input stream into an array of bytes.
Overrides:
`read` in class `InputStream`
Parameters:
`b` - the buffer into which the data is read
Returns:
the total number of bytes read into the buffer, or `-1` if there is no more data because the end of the file has been reached.
Throws:
`IOException` - if an I/O error occurs
`DFSException` - if a DFS specific error occurs
See Also:
`InputStream.read(byte[])`

reset

```
public final void reset()
                 throws IOException
```

Repositions this stream to the position at the time the `mark` method was last called on this input stream.

The general contract of `reset` is:

- If the method `markSupported` returns `true`, then:
  - If the method `mark` has not been called since the stream was created, or the number of bytes read from the stream since `mark` was last called is larger than the argument to `mark` at that last call, then an `IOException` might be thrown.
  - If such an `IOException` is not thrown, then the stream is reset to a state such that all the bytes read since the most recent call to `mark` (or since the start of the file, if `mark` has not been called) will be resupplied to subsequent callers of the `read` method, followed by any bytes that otherwise would have been the next input data as of the time of the call to `reset`.

Overrides:
`reset` in class `InputStream`
Throws:
`IOException` - if this stream has not been marked or if the mark has been invalidated
See Also:
`mark(int)`, `InputStream.reset()`, `IOException`

---

skip

```
public final long skip(long n)
                throws IOException
```

Skips over and discards `n` bytes of data from the input stream. The `skip` method may, for a variety of reasons, end up skipping over some smaller number of bytes, possibly `0`. The actual number of bytes skipped is returned.
Overrides:
`skip` in class `InputStream`
Parameters:
`n` - the number of bytes to be skipped
Returns:
the actual number of bytes skipped.
Throws:
`IOException` - if an I/O error occurs
See Also:
`InputStream.skip(long)`

Class DFSOutputStream

org.dbe.dfs

```
java.lang.Object
  └java.io.OutputStream
      └org.dbe.dfs.DFSOutputStream
```

public class **DFSOutputStream**
extends OutputStream

An OutputStream used to write data to a `DFSFile` , a file in the DFS.

Author:
Intel Ireland Ltd.
Version:
0.4.0

Field Summary	
private boolean	**closed** True if the stream is closed.
private String	**dfsFilePath** DFS path of the file accessed by this stream.
private org.dbe.dfs.service.DFSService	**dfsService** Reference to the DFSService implementation to be used.
private static final short	**EFF_THRESHOLD_EMPTY_BYTES** Constant that controls the mechanism for transmitting arrays efficiently.
private static final short	**EFF_THRESHOLD_NON_EMPTY_BYTES** Constant that controls the mechanism for transmitting arrays efficiently.

Constructor Summary	
public	**DFSOutputStream**(String name) Creates a new DFSOutputStream pointing to the supplied DFS path.
public	**DFSOutputStream**(String name, boolean append) Creates a new DFSOutputStream pointing to the supplied DFS path with the option that an existing file will be appended to.
protected	**DFSOutputStream**(String name, boolean append, org.dbe.dfs.service.DFSService service) Creates a new DFSOutputStream pointing to the supplied DFS path with the option that an existing file will be appended to, using the given DFS Service.
public	**DFSOutputStream**(DFSFile file) Creates a new DFSOutputStream pointing to the supplied DFSFile.
public	**DFSOutputStream**(DFSFile file, boolean append) Creates a new DFSOutputStream pointing to the supplied DFSFile with the option that an existing file will be appended to.

Method Summary	
final void	**close**() Closes this file output stream and releases any system resources associated with this stream.
final void	**flush**() Flushes this output stream and forces any buffered output bytes to be written out.

final void	**setTimeToLive**(long timeToLive) Sets the time-to-live for the file accessed by this stream.
final void	**write**(byte[] b) Writes b.length bytes from the specified byte array to this file output stream.
final void	**write**(byte[] b, int arrayOff, int len) Writes len bytes from the specified byte array starting at offset off to this file output stream.
final void	**write**(int b) Writes the specified byte to this file output stream.

Field Detail

dfsFilePath

```
private String dfsFilePath
```

DFS path of the file accessed by this stream.

---

dfsService

```
private org.dbe.dfs.service.DFSService dfsService
```

Reference to the DFSService implementation to be used.

---

closed

```
private boolean closed
```

True if the stream is closed. If true all the methods invoked on this object will fail.

EFF_THRESHOLD_EMPTY_BYTES

```
private static final short EFF_THRESHOLD_EMPTY_BYTES
```

Constant that controls the mechanism for transmitting arrays efficiently. If the number of empty transmitted bytes is less than this value, do not activate the mechanism.

EFF_THRESHOLD_NON_EMPTY_BYTES

```
private static final short EFF_THRESHOLD_NON_EMPTY_BYTES
```

Constant that controls the mechanism for transmitting arrays efficiently. If the number of non empty transmitted bytes is greater than this value, do not activate the mechanism.

Constructor Detail

DFSOutputStream

```
public DFSOutputStream(String name)
                throws FileNotFoundException,
                       DFSException
```

Creates a new DFSOutputStream pointing to the supplied DFS path.
Throws:
FileNotFoundException
DFSException

DFSOutputStream

```
public DFSOutputStream(String name,
                       boolean append)
                throws FileNotFoundException,
                       DFSException
```

Creates a new DFSOutputStream pointing to the supplied DFS path with the option that an existing file will be appended to.
Throws:
FileNotFoundException
DFSException

DFSOutputStream

```
public DFSOutputStream(DFSFile file)
                throws FileNotFoundException,
                       DFSException
```

Creates a new DFSOutputStream pointing to the supplied DFSFile.
Throws:
FileNotFoundException
DFSException

DFSOutputStream

```
public DFSOutputStream(DFSFile file,
                       boolean append)
                throws FileNotFoundException,
                       DFSException
```

Creates a new DFSOutputStream pointing to the supplied DFSFile with the option that an existing file will be appended to.
Throws:
FileNotFoundException
DFSException

DFSOutputStream

```
protected DFSOutputStream(String name,
                          boolean append,
                          org.dbe.dfs.service.DFSService service)
                   throws FileNotFoundException,
                          DFSException
```

Creates a new DFSOutputStream pointing to the supplied DFS path with the option that an existing file will be appended to, using the given DFS Service.

Throws:

`FileNotFoundException`
`DFSException`

Method Detail

setTimeToLive

```
public final void setTimeToLive(long timeToLive)
                          throws IOException,
                                 DFSException
```

Sets the time-to-live for the file accessed by this stream.

Parameters:

`timeToLive` - time-to-live for this file. It can not be negative.

Throws:

`IOException` - if a general I/O error occurs
`DFSException` - if a DFS specific error occurrs

---

write

```
public final void write(int b)
                 throws IOException,
                        DFSException
```

Writes the specified byte to this file output stream. Implements the `write` method of `OutputStream`.

Overrides:

`write` in class `OutputStream`

Parameters:

`b` - the byte to be written

Throws:

`IOException` - if an I/O error occurs
`DFSException` - if a DFS specific error occurrs

See Also:

`OutputStream.write(int)`

---

close

```
public final void close()
                 throws IOException
```

Closes this file output stream and releases any system resources associated with this stream. This file output stream may no longer be used for writing bytes.

Overrides:
`close` in class `OutputStream`
Throws:
`IOException` - if an I/O error occurs
See Also:
`OutputStream.close()`

---

flush

```
public final void flush()
                  throws IOException
```

Flushes this output stream and forces any buffered output bytes to be written out. The general contract of `flush` is that calling it is an indication that, if any bytes previously written have been buffered by the implementation of the output stream, such bytes should immediately be written to their intended destination. This implementation does not use buffering, and therefore this method does nothing.
Overrides:
`flush` in class `OutputStream`
Throws:
`IOException` - if an I/O error occurs
See Also:
`OutputStream.flush()`

---

write

```
public final void write(byte[] b,
                        int arrayOff,
                        int len)
                    throws IOException,
                        DFSException,
                        IllegalArgumentException,
                        ArrayIndexOutOfBoundsException
```

Writes `len` bytes from the specified byte array starting at offset `off` to this file output stream.
Overrides:
`write` in class `OutputStream`
Parameters:
`b` - the data
`arrayOff` - the start offset in the data array
`len` - the number of bytes to write
Throws:
`IOException` - if an I/O error occurs
`DFSException` - if a DFS specific error occurrs
`IllegalArgumentException` - if arrayOff or len are negative
`ArrayIndexOutOfBoundsException` - if the limits of the array b are exceeded
See Also:
`OutputStream.write(byte[], int, int)`

---

write

```
public final void write(byte[] b)
                 throws IOException,
                        DFSException
```

Writes `b.length` bytes from the specified byte array to this file output stream.

Overrides:

`write` in class `OutputStream`

Parameters:

`b` - the data

Throws:

`IOException` - if an I/O error occurs

`DFSException` - if a DFS specific error occurrs

See Also:

`OutputStream.write(byte[])`

Class DFSDirectoryInfo

org.dbe.dfs

```
java.lang.Object
  └org.dbe.dfs.DFSDirectoryInfo
```

All Implemented Interfaces:
Serializable

public class **DFSDirectoryInfo**
extends Object
implements Serializable

Contains information about the total number of files and the total amount of data contained in a directory, including all its subdirectories.

The number of files includes both the quantity of normal files and directories in a directory (and its subdirectories).

The number of bytes includes the size (in bytes) of all normal files in a directory (and its subdirectories). Thus an empty directory is considered to have a size of `0` bytes.

Author:
Intel Ireland Ltd.
Version:
0.4.0

Field Summary	
private long	**numBytes** Number of bytes occupied by files under the directory, including subdirectories.
private int	**numFiles** Number of files under the directory, including subdirectories.

Constructor Summary
DFSDirectoryInfo(int numberOfFiles, long numberOfBytes) Creates a new instance of this class.

Method Summary	
final void	**addBytes**(long number) Adds a given amount of bytes to the byte counter.
final void	**addFiles**(int number) Adds a given amount of files to the file counter.
final long	**getNumBytes**() Returns the total number of bytes of the files under this directory, including subdirectories.

final int	**getNumFiles**() Returns the total number of files under this directory, including subdirectories.

Field Detail

numFiles

```
private int numFiles
```

Number of files under the directory, including subdirectories.

---

numBytes

```
private long numBytes
```

Number of bytes occupied by files under the directory, including subdirectories.

Constructor Detail

DFSDirectoryInfo

```
public DFSDirectoryInfo(int numberOfFiles,
                        long numberOfBytes)
```

Creates a new instance of this class.

Method Detail

getNumBytes

```
public final long getNumBytes()
```

Returns the total number of bytes of the files under this directory, including subdirectories.
Returns:
the number of bytes.

---

getNumFiles

```
public final int getNumFiles()
```

Returns the total number of files under this directory, including subdirectories.
Returns:
the number of files.

---

addFiles

```
public final void addFiles(int number)
```

Adds a given amount of files to the file counter.
Parameters:
`number` - the number of files to be added

---

addBytes

```
public final void addBytes(long number)
```

Adds a given amount of bytes to the byte counter.
Parameters:
`number` - the number of bytes to be added

Class DFSFileInfo

org.dbe.dfs

```
java.lang.Object
  └org.dbe.dfs.DFSFileInfo
```

All Implemented Interfaces:
Serializable

---

public class **DFSFileInfo**
extends Object
implements Serializable

Contains information about a file in the DFS.

Author:
Intel Ireland Ltd.
Version:
0.4.0

Field Summary	
private boolean	**canRead** `true` if the file can be read.
private boolean	**canWrite** `true` if the file can be written.
private String	**dfsPath** Absolute path of the file.
private boolean	**isDirectory** `true` if the file is a directory.
private boolean	**isFile** `true` if the file is a regular file.
private long	**length** Length of the file, in bytes.
private String	**name** Name of the file (path relative to its parent directory)
private long	**timeToLive** 'Time to Live' for the file, in seconds.

Constructor Summary
DFSFileInfo(String path, String fileName, boolean isFileValue, boolean isDirectoryValue, boolean canReadValue, boolean canWriteValue, long lengthOfFile, long timeToLive) Creates a new DFSFileInfo object.

Method Summary	
final boolean	**canRead**() Returns the value of `canRead`.
final boolean	**canWrite**() Returns the value of `canWrite`.
final String	**getName**() Returns the name of the file.
final String	**getPath**() Returns the absolute path.
final boolean	**isDirectory**() Returns the value of `isDirectory`.
final boolean	**isFile**() Returns the value of `isFile`.
final long	**length**() Returns the length of the file.
protected final void	**setPath**(String path) Sets the path.
final long	**timeToLive**() Returns the timeToLive of the file.

Field Detail

dfsPath

```
private String dfsPath
```

Absolute path of the file.

---

name

```
private String name
```

Name of the file (path relative to its parent directory)

---

isFile

```
private boolean isFile
```

`true` if the file is a regular file.

---

isDirectory

```
private boolean isDirectory
```

`true` if the file is a directory.

---

canRead

```
private boolean canRead
```

`true` if the file can be read.

---

canWrite

```
private boolean canWrite
```

`true` if the file can be written.

---

length

```
private long length
```

Length of the file, in bytes.

---

timeToLive

```
private long timeToLive
```

'Time to Live' for the file, in seconds. Denotes how long the file should remain active

Constructor Detail

DFSFileInfo

```
public DFSFileInfo(String path,
                   String fileName,
                   boolean isFileValue,
                   boolean isDirectoryValue,
                   boolean canReadValue,
                   boolean canWriteValue,
                   long lengthOfFile,
                   long timeToLive)
```

Creates a new DFSFileInfo object.

Method Detail

canWrite

```
public final boolean canWrite()
```

Returns the value of `canWrite`.
Returns:
`true` if the file can be written.

---

getPath

```
public final String getPath()
```

Returns the absolute path.
Returns:
the absolute DFS file path.

---

isDirectory

```
public final boolean isDirectory()
```

Returns the value of `isDirectory`.
Returns:
`true` if the file is a directory.

---

isFile

```
public final boolean isFile()
```

Returns the value of `isFile`.
Returns:
`true` if the file is a regular file.

---

length

```
public final long length()
```

Returns the length of the file.
Returns:
length of the file, in bytes.

---

timeToLive

```
public final long timeToLive()
```

Returns the timeToLive of the file.
Returns:
timeToLive of the file second. Defines how long the file should exist in DBE

canRead

```
public final boolean canRead()
```

Returns the value of `canRead`.
Returns:
`true` if the file can be read.

---

getName

```
public final String getName()
```

Returns the name of the file.
Returns:
the name of the file, relative to its parent directory.

---

setPath

```
protected final void setPath(String path)
```

Sets the path.
Parameters:
`path` - the new path

B.2 DFS Service

Package Summary	
org.dbe.dfs.service	Provides generic interface and related classes required for a DBE File System (DFS) service.

Package org.dbe.dfs.service

Provides generic interface and related classes required for a DBE File System (DFS) service.

See:
Description

Interface Summary	
DFSService	Describes an interface for a service which implements a DBE File System.

Class Summary	
DFSServiceReadResult	Encapsulates the result of reading an array of bytes.

Exception Summary	
DFSRemoteException	Signals a communications-specific exception has occurred.

Package org.dbe.dfs.service Description

Provides generic interface and related classes required for a DBE File System (DFS) service.

Author:

Intel Ireland Ltd.

Version:
0.4

Class DFSRemoteException

org.dbe.dfs.service

```
java.lang.Object
  └java.lang.Throwable
      └java.lang.Exception
          └org.dbe.dfs.service.DFSRemoteException
```

All Implemented Interfaces:
Serializable

public class **DFSRemoteException**
extends Exception

Signals a communications-specific exception has occurred.

Author:
Intel Ireland Ltd.
Version:
0.4.0

Constructor Summary
DFSRemoteException() Creates a new exception.
DFSRemoteException(String message) Creates a new exception with the given message.
DFSRemoteException(String message, Throwable cause) Creates a new exception with the given message and cause.
DFSRemoteException(Throwable cause) Creates a new exception with the given message and cause.

Constructor Detail

DFSRemoteException

```
public DFSRemoteException(String message,
                          Throwable cause)
```

Creates a new exception with the given message and cause.

DFSRemoteException

```
public DFSRemoteException(Throwable cause)
```

Creates a new exception with the given message and cause.

DFSRemoteException

```
public DFSRemoteException()
```

Creates a new exception.

DFSRemoteException

```
public DFSRemoteException(String message)
```

Creates a new exception with the given message.

Interface DFSService

org.dbe.dfs.service

All Superinterfaces:
Serializable

---

public interface **DFSService**
extends Serializable

Describes an interface for a service which implements a DBE File System.
Whilst pathnames supplied to the DFSService may use '/', '//', '\' or '\\' as the path separator, all DFSService methods that return paths (for example list() and listFiles()) will use the separatorChar as the path separator.

Author:
Intel Ireland Ltd.
Version:
0.4.0

Field Summary	
String	**separator** The default name-separator character used by this instance of the DFSService, represented as a string for convenience.
char	**separatorChar** The default name-separator character used by this instance of the DFSService.

Method Summary	
boolean	**canRead**(String dfsPath) Tests whether the application can read the file denoted by the abstract pathname supplied.
boolean	**canWrite**(String dfsPath) Tests whether the application can write to the file denoted by the abstract pathname supplied.
boolean	**createNewFile**(String dfsPath, long timeToLive) Atomically creates a new, empty file named by the abstract pathname supplied, if and only if a file with this name does not yet exist.
boolean	**delete**(String dfsPath) Deletes the file or directory denoted by the abstract pathname supplied.
DFSDirectoryInfo	**directoryInfo**(String dfsPath) Returns the total number of files and the total amount of bytes of this directory, including (recursively) all its subdirectories.
boolean	**exists**(String dfsPath) Tests whether the file or directory denoted by the abstract pathname supplied exists.

long	**getTimeToLive**(String dfsPath) Returns the number of seconds remaining before the file specified in the abstract pathname supplied may be automatically deleted from the DFS.
boolean	**isDirectory**(String dfsPath) Tests whether the file denoted by the abstract pathname supplied is a directory.
boolean	**isFile**(String dfsPath) Tests whether the file denoted by the abstract pathname supplied is a normal file.
long	**length**(String dfsPath) Returns the length of the file denoted by the abstract pathname supplied.
String[]	**list**(String dfsPath) Returns an array of strings naming the files and directories in the directory denoted by the abstract pathname supplied.
String[]	**listFiles**(String dfsPath) Returns an array of abstract pathnames denoting the files and directories in the directory denoted by the abstract pathname supplied.
DFSFileInfo[]	**listFilesInfo**(String dfsPath) Returns an array of information about the files and directories in the directory denoted by the abstract pathname supplied.
boolean	**mkdir**(String dfsPath) Creates the directory named by the abstract pathname supplied.
boolean	**mkdirs**(String dfsPath) Creates the directory named by the abstract pathname supplied, including any necessary but nonexistent parent directories.
int	**read**(String dfsPath, long fileOff) Reads a byte of data from the abstract pathname supplied.
DFSServiceReadResult	**read**(String dfsPath, long fileOff, int len) Reads up to `len` bytes of data from the abstract pathname supplied, starting at the `off` position in that file.
boolean	**renameTo**(String fromDFSPath, String toDFSPath) Renames the file denoted by the first abstract pathname supplied to that denoted by the second abstract pathname supplied.
boolean	**setTimeToLive**(String dfsPath, long timeToLive) Assigns a time-to-live value, in seconds, to the file denoted by the abstract pathname supplied.
void	**write**(String dfsPath, byte[] b) Writes b.length bytes from the specified byte array to the abstract pathname supplied.
void	**write**(String dfsPath, byte[] b, boolean append) Writes `b.length` bytes from the specified byte array to the abstract pathname supplied.
void	**write**(String dfsPath, byte[] b, int arrayOff, int len) Writes len bytes from the specified byte array, starting at offset off, to the abstract pathname supplied.
void	**write**(String dfsPath, byte[] b, int arrayOff, int len, boolean append) Writes len bytes from the specified byte array, starting at offset off, to the abstract pathname supplied.
void	**write**(String dfsPath, int b) Writes the specified byte to the abstract pathname supplied.
void	**write**(String dfsPath, int b, boolean append) Writes the specified byte to the abstract pathname supplied.

Field Detail

separator

```
public static final String separator
```

The default name-separator character used by this instance of the DFSService, represented as a string for convenience. This string contains a single character, namely separatorChar.

---

separatorChar

```
public static final char separatorChar
```

The default name-separator character used by this instance of the DFSService. For the DFS, this character is '/'.

Method Detail

createNewFile

```
public boolean createNewFile(String dfsPath,
                             long timeToLive)
                      throws IOException,
                             DFSRemoteException
```

Atomically creates a new, empty file named by the abstract pathname supplied, if and only if a file with this name does not yet exist. The check for the existence of the file and the creation of the file if it does not exist are a single operation that is atomic with respect to all other filesystem activities that might affect the file.

Parameters:

`dfsPath` - the abstract DFS path

`timeToLive` - the amount of time, in seconds, after which the file may be automatically deleted from the DFS. It must be greater than 0.

Returns:

`true` if the named file does not exist and was successfully created; `false` if the named file already exists.

Throws:

`IOException` - if an I/O error occurs

`DFSRemoteException` - if an error occurs during the invocation of the remote service

---

exists

```
public boolean exists(String dfsPath)
               throws DFSRemoteException,
                      IOException
```

Tests whether the file or directory denoted by the abstract pathname supplied exists.

Parameters:

`dfsPath` - the abstract DFS path

Returns:

`true` if and only if the file or directory denoted by this abstract pathname exists; `false` otherwise.

Throws:

DFSRemoteException - if an error occurs during the invocation of the remote service

IOException - if an I/O error occurs

---

delete

```
public boolean delete(String dfsPath)
               throws DFSRemoteException,
                      IOException
```

Deletes the file or directory denoted by the abstract pathname supplied. If this pathname denotes a directory, then the directory must be empty in order to be deleted.

Parameters:

dfsPath - the abstract DFS path

Returns:

true if and only if the file or directory is successfully deleted; false otherwise.

Throws:

DFSRemoteException - if an error occurs during the invocation of the remote service

IOException - if an I/O error occurs

---

length

```
public long length(String dfsPath)
            throws DFSRemoteException,
                   IOException
```

Returns the length of the file denoted by the abstract pathname supplied. The return value is unspecified if this pathname denotes a directory.

Parameters:

dfsPath - the abstract DFS path

Returns:

the length, in bytes, of the file denoted by the abstract pathname supplied, or 0L if the file does not exist.

Throws:

DFSRemoteException - if an error occurs during the invocation of the remote service

IOException - if an I/O error occurs

---

renameTo

```
public boolean renameTo(String fromDFSPath,
                        String toDFSPath)
                 throws DFSRemoteException,
                        IOException
```

Renames the file denoted by the first abstract pathname supplied to that denoted by the second abstract pathname supplied. The return value should always be checked to make sure that the rename operation was successful.

Parameters:

fromDFSPath - the existing abstract DFS path

toDFSPath - the new abstract DFS path

Returns:

`true` if and only if the renaming succeeded; `false` otherwise.

Throws:

`DFSRemoteException` - if an error occurs during the invocation of the remote service

`IOException` - if an I/O error occurs

---

canRead

```
public boolean canRead(String dfsPath)
                throws DFSRemoteException,
                       IOException
```

Tests whether the application can read the file denoted by the abstract pathname supplied.

Parameters:

`dfsPath` - the abstract DFS path

Returns:

`true` if and only if the file specified by the abstract pathname supplied exists and can be read by the application; `false` otherwise.

Throws:

`DFSRemoteException` - if an error occurs during the invocation of the remote service

`IOException` - if an I/O error occurs

---

canWrite

```
public boolean canWrite(String dfsPath)
                 throws DFSRemoteException,
                        IOException
```

Tests whether the application can write to the file denoted by the abstract pathname supplied.

Parameters:

`dfsPath` - the abstract DFS path

Returns:

`true` if and only if the file system actually contains a file denoted by the abstract pathname supplied and the application is allowed to write to the file; `false` otherwise.

Throws:

`DFSRemoteException` - if an error occurs during the invocation of the remote service

`IOException` - if an I/O error occurs

---

mkdir

```
public boolean mkdir(String dfsPath)
              throws DFSRemoteException,
                     IOException
```

Creates the directory named by the abstract pathname supplied.

Parameters:

`dfsPath` - the abstract DFS path

Returns:

`true` if and only if the directory was created; `false` otherwise.

Throws:

DFSRemoteException - if an error occurs during the invocation of the remote service

IOException - if an I/O error occurs

---

mkdirs

```
public boolean mkdirs(String dfsPath)
               throws DFSRemoteException,
                      IOException
```

Creates the directory named by the abstract pathname supplied, including any necessary but nonexistent parent directories. Note that if this operation fails it may have succeeded in creating some of the necessary parent directories.

Parameters:

dfsPath - the abstract DFS path

Returns:

true if and only if the directory was created, along with all necessary parent directories; false otherwise.

Throws:

DFSRemoteException - if an error occurs during the invocation of the remote service

IOException - if an I/O error occurs

---

isFile

```
public boolean isFile(String dfsPath)
               throws DFSRemoteException,
                      IOException
```

Tests whether the file denoted by the abstract pathname supplied is a normal file. A file is normal in the DFS if it is not a directory.

Parameters:

dfsPath - the abstract DFS path

Returns:

true if and only if the file denoted by the abstract pathname supplied exists and is a normal file; false otherwise.

Throws:

DFSRemoteException - if an error occurs during the invocation of the remote service

IOException - if an I/O error occurs

---

isDirectory

```
public boolean isDirectory(String dfsPath)
                    throws DFSRemoteException,
                           IOException
```

Tests whether the file denoted by the abstract pathname supplied is a directory.

Parameters:

dfsPath - the abstract DFS path

Returns:

`true` if and only if the file denoted by the abstract pathname supplied exists and is a directory; `false` otherwise.

Throws:

`DFSRemoteException` - if an error occurs during the invocation of the remote service

`IOException` - if an I/O error occurs

---

list

```
public String[] list(String dfsPath)
             throws DFSRemoteException,
                    IOException
```

Returns an array of strings naming the files and directories in the directory denoted by the abstract pathname supplied. If the abstract pathname supplied does not denote a directory, then this method returns `null`. Otherwise an array of strings is returned, one for each file or directory in the directory. Names denoting the directory itself and the directory's parent directory are not included in the result. Each string is a file or directory name rather than an absolute path. There is no guarantee that the name strings in the resulting array will appear in any specific order; they are not, in particular, guaranteed to appear in alphabetical order.

Parameters:

`dfsPath` - the abstract DFS path

Returns:

an array of strings naming the files and directories in the directory denoted by the abstract pathname supplied. The array will be empty if the directory is empty. Returns `null` if the abstract pathname supplied does not denote a directory, or if an I/O error occurs.

Throws:

`DFSRemoteException` - if an error occurs during the invocation of the remote service

`IOException` - if an I/O error occurs

---

listFiles

```
public String[] listFiles(String dfsPath)
                 throws DFSRemoteException,
                        IOException
```

Returns an array of abstract pathnames denoting the files and directories in the directory denoted by the abstract pathname supplied. If the abstract pathname supplied does not denote a directory, then this method returns `null`. Otherwise an array of strings is returned, one for each file or directory in the directory. Pathnames denoting the directory itself and the directory's parent directory are not included in the result. Each string is an absolute path. There is no guarantee that the name strings in the resulting array will appear in any specific order; they are not, in particular, guaranteed to appear in alphabetical order.

Parameters:

`dfsPath` - the abstract DFS path

Returns:

an array of strings of absolute pathnames denoting the files and directories in the directory denoted by the abstract pathname supplied. The array will be empty if the directory is empty. Returns `null` if the abstract pathname supplied does not denote a directory, or if an I/O error occurs.

Throws:

`DFSRemoteException` - if an error occurs during the invocation of the remote service

`IOException` - if an I/O error occurs

listFilesInfo

```
public DFSFileInfo[] listFilesInfo(String dfsPath)
                            throws DFSRemoteException,
                                   IOException
```

Returns an array of information about the files and directories in the directory denoted by the abstract pathname supplied. If the abstract pathname supplied does not denote a directory, then this method returns `null`. Otherwise an array of DFSFileInfo is returned, one for each file or directory in the directory. There is no guarantee that the name strings in the resulting array will appear in any specific order; they are not, in particular, guaranteed to appear in alphabetical order.

Parameters:

`dfsPath` - the abstract path of a directory

Returns:

an array with information about all the files, or `null` if the abstract pathname does not denote a directory.

Throws:

`DFSRemoteException` - if an error occurs during the invocation of the remote service

`IOException` - if an I/O error occurs

---

setTimeToLive

```
public boolean setTimeToLive(String dfsPath,
                             long timeToLive)
                      throws DFSRemoteException,
                             FileNotFoundException,
                             IOException
```

Assigns a time-to-live value, in seconds, to the file denoted by the abstract pathname supplied. Time-to-live does not apply to directories.

Parameters:

`dfsPath` - the abstract DFS path

`timeToLive` - the amount of time, in seconds, after which the file may be automatically deleted from the DFS

Returns:

`true` if the abstract pathname supplied is not a directory, and the time-to-live value is successfully applied to it; `false` otherwise.

Throws:

`DFSRemoteException` - if an error occurs during the invocation of the remote service

`FileNotFoundException` - if the abstract pathname does not exist

`IOException` - if an I/O error occurs

---

getTimeToLive

```
public long getTimeToLive(String dfsPath)
                   throws DFSRemoteException,
                          FileNotFoundException,
                          IOException
```

Returns the number of seconds remaining before the file specified in the abstract pathname supplied may be automatically deleted from the DFS.

Parameters:
`dfsPath` - the abstract DFS path
Returns:
the amount of time, in seconds, after which the file may be automatically deleted from the DFS, or `-1` if the abstract DFS path points to a directory. All directories have an infinite time-to-live.
Throws:
`DFSRemoteException` - if an error occurs during the invocation of the remote service
`FileNotFoundException` - if the abstract pathname does not exist
`IOException` - if an I/O error occurs

---

read

```
public DFSServiceReadResult read(String dfsPath,
                                 long fileOff,
                                 int len)
                          throws IOException,
                                 DFSRemoteException
```

Reads up to `len` bytes of data from the abstract pathname supplied, starting at the `off` position in that file. This method blocks until some input is available.
Parameters:
`dfsPath` - the abstract DFS path
`fileOff` - the start offset of the data
`len` - the maximum number of bytes to be read
Returns:
an object encapsulating the number of bytes read and an array of bytes containing the bytes read.
Throws:
`IOException` - if an I/O error occurs
`DFSRemoteException` - if an error occurs during the invocation of the remote service

---

read

```
public int read(String dfsPath,
                long fileOff)
         throws IOException,
                DFSRemoteException
```

Reads a byte of data from the abstract pathname supplied. This method blocks until some input is available.
Parameters:
`dfsPath` - the abstract DFS path
`fileOff` - the start offset of the data
Returns:
the byte of data read, or `-1` if the end of the file is reached.
Throws:
`IOException` - if an I/O error occurs
`DFSRemoteException` - if an error occurs during the invocation of the remote service

---

write

```
public void write(String dfsPath,
                  int b)
           throws IOException,
                  DFSRemoteException
```

Writes the specified byte to the abstract pathname supplied. If the file already exists, the byte is appended to the file. If not, the file is created, subject to the limitations of the CreateNewFile() method.

Parameters:

`dfsPath` - the abstract DFS path

`b` - the byte to be written

Throws:

`IOException` - if an I/O error occurs

`DFSRemoteException` - if an error occurs during the invocation of the remote service

---

write

```
public void write(String dfsPath,
                  byte[] b)
           throws IOException,
                  DFSRemoteException
```

Writes b.length bytes from the specified byte array to the abstract pathname supplied. If the file already exists, the bytes are appended to the file. If not, the file is created, subject to the limitations of the CreateNewFile() method.

Parameters:

`dfsPath` - the abstract DFS path

`b` - the data

Throws:

`IOException` - if an I/O error occurs

`DFSRemoteException` - if an error occurs during the invocation of the remote service

---

write

```
public void write(String dfsPath,
                  byte[] b,
                  int arrayOff,
                  int len)
           throws IOException,
                  DFSRemoteException
```

Writes len bytes from the specified byte array, starting at offset off, to the abstract pathname supplied. If the file already exists, the bytes are appended to the file. If not, the file is created, subject to the limitations of the CreateNewFile() method.

Parameters:

`dfsPath` - the abstract DFS path

`b` - the data.

`arrayOff` - the start offset in the data

`len` - the number of bytes to write

Throws:

`IOException` - if an I/O error occurs

`DFSRemoteException` - if an error occurs during the invocation of the remote service

write

```
public void write(String dfsPath,
                  int b,
                  boolean append)
           throws IOException,
                  DFSRemoteException
```

Writes the specified byte to the abstract pathname supplied. If the file already exists, the byte is appended to the file or written at the beginning of the file. If not, the file is created, subject to the limitations of the CreateNewFile() method.

Parameters:

`dfsPath` - the abstract DFS path

`b` - the byte to be written

`append` - `true` if data should be appended or `false` if it should be written at the beginning of the file

Throws:

`IOException` - if an I/O error occurs

`DFSRemoteException` - if an error occurs during the invocation of the remote service

---

write

```
public void write(String dfsPath,
                  byte[] b,
                  boolean append)
           throws IOException,
                  DFSRemoteException
```

Writes `b.length` bytes from the specified byte array to the abstract pathname supplied. If the file already exists, the bytes are appended to the file or written at the beginning of the file. If not, the file is created, subject to the limitations of the CreateNewFile() method.

Parameters:

`dfsPath` - the abstract DFS path

`b` - the data

`append` - `true` if data should be appended or `false` if it should be written at the beginning of the file

Throws:

`IOException` - if an I/O error occurs

`DFSRemoteException` - if an error occurs during the invocation of the remote service

---

write

```
public void write(String dfsPath,
                  byte[] b,
                  int arrayOff,
                  int len,
                  boolean append)
           throws IOException,
                  DFSRemoteException
```

Writes len bytes from the specified byte array, starting at offset off, to the abstract pathname supplied. If the file already exists, the bytes are appended to the file or written at the beginning of the file. If not, the file is created, subject to the limitations of the CreateNewFile() method.

Parameters:

`dfsPath` - the abstract DFS path

`b` - the data

`arrayOff` - the start offset in the data

`len` - the number of bytes to write

`append` - `true` if data should be appended or `false` if it should be written at the beginning of the file

Throws:

`IOException` - if an I/O error occurs

`DFSRemoteException` - if an error occurs during the invocation of the remote service

---

directoryInfo

```
public DFSDirectoryInfo directoryInfo(String dfsPath)
                              throws IOException,
                                     DFSRemoteException
```

Returns the total number of files and the total amount of bytes of this directory, including (recursively) all its subdirectories.

Parameters:

`dfsPath` - the abstract path of the directory

Returns:

the information about this directory, or `null` if the DFS path is not a directory.

Throws:

`IOException` - if an I/O error occurs

`DFSRemoteException` - if an error occurs during the invocation of the remote service

Class DFSServiceReadResult

org.dbe.dfs.service

```
java.lang.Object
  └org.dbe.dfs.service.DFSServiceReadResult
```

All Implemented Interfaces:
Serializable

---

public class **DFSServiceReadResult**
extends Object
implements Serializable

Encapsulates the result of reading an array of bytes. The result consists of the number of bytes read (or `-1` if End-of-File reached), and an array with the bytes read.

Author:
Intel Ireland Ltd.
Version:
0.4.0

Field Summary	
protected byte[]	**bytesRead** The bytes read.
protected int	**numBytesRead** The number of bytes read (or -1 if EOF was reached and no bytes were read).

Constructor Summary
DFSServiceReadResult(int numBytesReadP, byte[] bytesReadP) Creates a new object with the given attributes.

Method Summary	
final byte[]	**getBytesRead**() Returns the array of bytes read.
final int	**getNumBytesRead**() Returns the number of bytes read.

Field Detail

numBytesRead

```
protected int numBytesRead
```

The number of bytes read (or -1 if EOF was reached and no bytes were read).

bytesRead

```
protected byte[] bytesRead
```

The bytes read.

Constructor Detail

DFSServiceReadResult

```
public DFSServiceReadResult(int numBytesReadP,
                            byte[] bytesReadP)
```

Creates a new object with the given attributes.

Method Detail

getBytesRead

```
public final byte[] getBytesRead()
```

Returns the array of bytes read.
Returns:
the array of bytes read.

---

getNumBytesRead

```
public final int getNumBytesRead()
```

Returns the number of bytes read.
Returns:
the number of bytes read (-1 if EOF was reached and no bytes were read)

Appendix C DSS Deployment.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<deploymentProperties xmlns="http://dbestudio.sourceforge.net/schemas/service-exporter/">
  <serviceInfo>
    <serviceName>DSSService</serviceName>
    <shortDescription>DSS Service</shortDescription>
    <adapterClass>org.dbe.dss.service.DSSServiceImpl</adapterClass>
  </serviceInfo>

  <entries>
    <entry>DSS</entry>
    <entry>DSS-c37f27cb-4acb-4fed-9047-7be9879cbfaa</entry>
  </entries>


<!-- DSS Store settings -->
  <property>
    <name>dss.store.type</name>
    <value>org.dbe.dss.service.store.localdir.LocalDirStoreConnector</value>
  </property>
  <property>
    <name>dss.store.id</name>
    <value>DSS-c37f27cb-4acb-4fed-9047-7be9879cbfaa</value>
  </property>
<!-- Uncomment to specify the absolute location of the dss store directory
  <property>
    <name>dss.store.directory</name>
    <value>C:\\DBE\\dss</value>
  </property>
-->

<!-- DSS Index settings -->
  <property>
    <name>dss.index.type</name>
    <value>org.dbe.dss.service.index.service.ServiceIndexConnector</value>
<!--   <value>org.dbe.dss.service.index.postgresql.PostgreSQLIndexConnector</value>  -->
  </property>

<!-- Uncomment the following four properties if using the PostgreSQL index
  <property>
    <name>dss.index.type</name>
  </property>
  <property>
    <name>dss.index.url</name>
    <value>jdbc:postgresql://localhost/dss_index</value>
  </property>
  <property>
    <name>dss.index.user</name>
    <value>dss_index</value>
  </property>
  <property>
    <value>dss.index.password</value>
    <name>password goes here </name>
  </property>
-->
</deploymentProperties>
```

Appendix D DFS Deployment.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<deploymentProperties xmlns="http://dbestudio.sourceforge.net/schemas/service-exporter/">
  <serviceInfo>
    <serviceName>DFSService</serviceName>
    <shortDescription>DFS Service</shortDescription>
    <adapterClass>org.dbe.dfs.service.basicfs.BasicFSServiceImpl</adapterClass>
  </serviceInfo>
  <entries>
    <entry>DFS</entry>
    <entry>DFS-JMKENNED</entry>
  </entries>

<!-- DFS Instance settings -->
  <property>
    <name>dfs.instance.id</name>
    <value>DFS-JMKENNED</value>
  </property>

<!-- DFS Index settings -->
  <property>
    <name>dfs.index.type</name>
    <value>org.dbe.dfs.service.basicfs.index.postgresql.PostgreSQLIndexConnector</value>
  </property>
  <property>
    <name>dfs.index.url</name>
    <value>jdbc:postgresql://localhost/bfs</value>
  </property>
  <property>
    <name>dfs.index.user</name>
    <value>bfs_service</value>
  </property>
  <property>
    <value>dfs.index.password</value>
    <name>password goes here</name>
  </property>
</deploymentProperties>
```