The Design of a Biologically Inspired Peer-to-Peer Distributed File System

Sergio Camorlinga[1] and Ken Barker[1&2]
[1]Department of Computer Science, Univ. of Manitoba, Winnipeg, MB Canada
[2]Department of Computer Science, Univ. of Calgary, Calgary, AB Canada

1. Introduction

This paper introduces the design of a *Biologically Inspired Peer-to-Peer Distributed File System* (BPD). *Peer-to-Peer* (P2P) computing is characterized by a dynamic, self-organization, ad-hoc connectivity of its decentralized members with challenges in terms of security, performance, interoperability and fault resilience among others [1]. *Complex Adaptive Systems* (CAS) are characterized by having a large number of members with simple functions and limited communications among them. The swarm intelligence [2] that emerges from the global behaviors of simple members boasts autonomy and self-sufficiency, which allows them to adapt quickly to changing environmental conditions. Common CAS member behaviors are based on biological and natural systems where team or group models solve complex distributed problems (e.g. insect and animal societies). Our BPD design merges CAS models with P2P computing to implement a *Distributed File System* (DFS). The BPD design presents a novel alternative to existing deterministic, centralized or distributed techniques proposed in the majority of current P2P and DFS research.

The BPD is designed with the capability to adapt, scale and perform in a web of computing devices. The BPD assumes an environment where you have computing

devices with ad hoc behavior (i.e. joining and leaving the network) and no central server or controller. The BPD is intended to scale from few peers to hundred of peers (probably thousands) allocated across a distributed system within an organization.

Medium and large organizations have hundreds of employees with personal computers (PCs) located within several branches that probably are physically apart from each other (e.g. large financial and management companies). The BPD targets this environment where users can seamlessly and dynamically share their storage resources to provide a peer-to-peer (P2P) distributed file system (DFS). This P2P DFS is used for the employees' applications and document management needs. For example, if the average user, in a 1000-employee organization, has available 50 GB spare space in his/her PC, then the BPD potentially can create a 50 TB P2P DFS. BPD is built from a set of commodity peers available across the enterprise that participates sharing their resources. Since these peers can fail at any moment, the BPD adaptability and fault tolerance by means of CAS models take the necessary steps to maintain the reliability on data availability.

A clear departure of our BPD from previously reported P2P DFS is that fact that application semantics is supported. The supported semantics relates to the way data is created and later retrieved, specifically data hierarchies (application dependencies) and data relationships (proximity within the hierarchy). There are many examples in our everyday environments where data semantics exists and are used when applications

access data. For instance, in a large insurance company, the majority of services and products go around the customer data and the customer's insurance portfolio (e.g. customers and their insurance personal belongings like house, car, health, *etc.*). There exists a clear relationship and hierarchy among related data. Another example can be a regional hospital facility where services provided goes around patients and their medical records (e.g. patient health records with lab results, radiology images, prescriptions, diagnosis and treatments, bills, *etc.*). Data hierarchies and relationships exist in every organization. By supporting data semantics, the BPD permeates application domain context into the DFS data management.

Data hierarchies and relationships are created and retrieved by data producers and consumers potentially located in many different places across an organization. This situation creates a physical fragmentation of related data in different, probably distant, locations. Even though data is fragmented in several storage islands, data semantic hierarchies and relationships continue to exist and should be supported for a more efficient data retrieval and storage. For this, the BPD introduces the concept of a *bag*, where a *bag* is a hierarchy of related data. Furthermore the BPD extends standard File Systems (FS) Application Programming Interface (API) to support bag semantics and provide better access to related data and their data relationships.

Contrary to a conventional hierarchical FS with a common global root usually found in other DFS designs [3], The BPD has a floating multi root hierarchical FS. Roots are

defined according to application semantics and are dynamically created/expanded/deleted as required by the client application to form a *bag*. There could be several directory hierarchies (i.e. *bags*) per client in a given time. This supports data independence and relationships that exists in application data. For this, the concept of *bag holder* is also introduced that dynamically holds several directory hierarchies or *bags*.

A common file life cycle follows continuous updates for a relatively short period of time to be either deleted [15] or stored. Once the file is stored, the file has few or no changes for some time. During this period of time the file has several read retrievals until eventually it is backed-up or deleted. The BPD takes into account this file life cycle behavior to manage data more efficiently. Also, the BPD has policies to keep locally the data for immediate access. After the client closes the file, then the BPD allocates the file in the distributed system.

Section 2 describes related work to our BPD particularly recent P2P DFS research work and semantically archive management systems. Section 3 gives a design overview of the BPD architecture to continue in section 4, where the architecture stack is described in detail. Section 5 describes protocols that glue the layers in the architecture. Additional information for data management is also included in this section. Section 6 briefly describes current and future work to conclude in section 7 by stressing some of the BPD features.

2. Related Work

DFS is a research area where many interesting work has been produced since late 1970's and early 1980's when seminal ideas were developed. Nowadays this trend continues where recent peer-to-peer computing research has been merge with DFS, producing a flurry of research work. The BPD expands this work with CAS computing models. CAS computing provides to a P2P DFS implicit adaptability, fault tolerance and scalability, which makes the BPD, differentiate from other previous reported research work. Consequently, this section focuses discussion in recent related P2P DFS work that is relevant to our work.

The BPD similar to GFS [3] consists of a P2P community of commodity devices whose quality and quantity guarantee that some of them will fail at any given time. However the BPD automatically adapts to failure by means of CAS models instead of a constant monitoring, error detection, fault tolerance and recovery pre-designed in GFS. The BPD peers participate in the DFS according to their capabilities and logically are members of a community; contrary with GFS that has a master server with chunk servers as storage slaves for GFS clients.

A commonality of some previous work is the use of a lower layer lookup service to store data across a set of servers (e.g. CFS [6] uses Chord [8], PAST [7] uses Pastry [9], Oceanstore [11] uses Tapestry [12], *etc*). These lookup services guarantee a lookup time

in the number of hops, however they lose the data locality. The data locality lost can bring latency in data storage and retrieval [10], unless special workarounds are designed. The BPD uses a different approach by keeping the metadata locally in each peer and using CAS computing layer services to store, retrieve and search data with implicitly high data locality.

DFSs like Frangipani [13], xFS [14] and others use caching mechanisms to offer global file cache. The BPD does not have any global cache, since relies on commodity local FS caches to provide some level of local caching. Also the BPD has a policy to keep locally the data to minimize access delays. After the client closes the file, then the BPD allocates the file in the distributed system. This policy helps the BPD to follow the life cycle of many file changes made by the client before saving permanently the file.

Data management is done dynamically at the clients with a clear separation of storage from file management similar to zFS [24], Elliot [25] and others. However the BPD is simpler and faster, as each client does its data management locally and independently. Furthermore, each client dynamically does data management when building remote bags of related data that it managed by other clients.

Semantic file systems (SFS) [17] and data archive management systems [16][18][19] have associated metadata via attributes-values to files similar to the BPD to support semantic data access. SFS and data archive management systems have servers that are

responsible to catalog indexes, manage directory trees and provide search capabilities, which contrary to the BPD where this functionality is spread out across the P2P system, and achieved by emergent search results coming out of the collaborative CAS model applied.

3. Design Overview

3.1 Goals

The BPD has several goals in its design, particularly:

- it should run in an environment that exists in a typical organization with PCs sharing their storage resources,
- it should have adaptability and scalability to flexible environments due to ad-hoc peer operation,
- it should balance resource usage across the distributed system,
- it should support application domain semantics, specifically data relationships and hierarchies, and
- it should support spatial and temporal data locality to provide a reasonable data retrieval and storage latency.

3.2 Assumptions

The following assumptions have been used for BPD design.

- Commodity peers (i.e. PCs, storage devices, servers) forming a BPD peer network. These peers can fail at any moment.

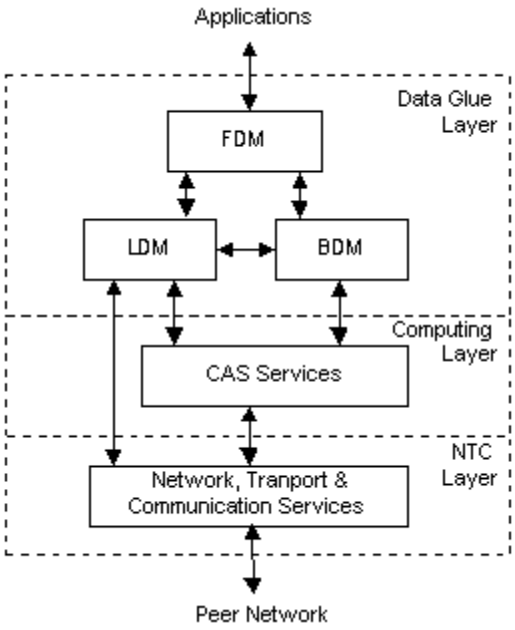
- Relatively loosely-coupled peers, joining and leaving the BPD web at any time.
- A client can have multiple directory hierarchies. These hierarchies are movable and sharable across clients and follow a semantic structure provided by the application domain.
- Data consists of many files, which are related and hierarchical structured per some semantic meaning specified by the application domain. Thr related data structure could become a directory with its own root.
- Common files follow a generic life cycle of continuous updates for a relatively short period of time to be either deleted or stored. Once the file is stored, the file has few or no changes.
- Replication is used to provide an acceptable file storage reliability level.

3.3 BPD Architecture Overview

The BPD architecture stack (Figure 1) runs in every machine that participates in the web of peers forming the distributed file system. The architecture stack is divided in several layers that together provide file system services to applications running on the peer. These layers are:

- Data glue layer responsible to provide metadata services to keep control of files and directories. It consists of Front-end Data Manager (FDM), Local-end Data Manager (LDM) and Back-end Data Manager (BDM). The data glue layer also has the Application Programming Interface (API), the Common Line Interface (CLI) and the Graphical User Interface (GUI) interfaces to client applications.

- Computing layer provides the file system adaptability by using Complex Adaptive System (CAS) models and algorithms to provide allocation, retrieval and search data services.
- Network, Transport and Communication (NTC) layer interconnects peers across the distributed system with networking and communication services. It interfaces to the physical interconnected peer network.



BPD Architecture Stack
Figure 1

BPD introduces the concept of a *bag* to represent different containers where related data is kept. A *bag* is defined as a complete directory hierarchy of related files. The implicit locality of the CAS models used by the BPD will automatically localize files that are under the same root directory hierarchy or *bag*. In this way, data locality is implemented by the use of *bags*. To manage the *bag*, the concept of a *bag item* (or *item* for simplicity)

is introduced. An *item* is defined as a file, a folder within a bag or the bag itself. All FS function service calls are defined around the *bag item* concept (see *API Interface* section).

A *bags' holder* (or *holder* for simplicity) is a virtual place where multiple bags can dynamically exist. There is one *holder* per client computer. A *holder* grows as more bags are loaded, and shrinks as bags are unloaded. Bags are loaded automatically by changing directory to a different root folder not already loaded. Bags are unloaded automatically following a least recently used scheme for bags currently residing in the holder. A holder has a limit in the number of bags capable to keep. Loading and unloading bags will implement the BPD floating multi root hierarchical FS behavior, i.e. the client execution pattern is driving the holder's behavior dynamics.

4. Design Architecture

4.1 API Interface

BPD application programming interface (API) is similar to regular APIs provided by other file system (FS). The BPD API is not POSIX (Portable Operating System Interface) compliant [4]. However, the BPD supports equivalent standard FS APIs like create, delete, open, close, read, write, *etc*. This is because the BPD research is focused on the application of CAS models in a P2P DFS and its benefits, rather than offering a FS with POSIX [5] conformance. Standard FS APIs function calls are renamed and/or extended to reflect a more appropriate functionality to BPD services and concepts.

The BPD API interface includes the following system calls:

- itemId openItem (char *path, int oFlag)

  Opens an existing item (i.e. bag item) and/or create an item if it does not exist. Returns an item identifier that links to the opened/created item. The value of oFlag sets up the item flags and item access modes. If the item being created is not a file (indicated in item flags), then there are two scenarios:

  - If path starts with '/' and there is only one path item and the path item name does not exist, then a bag is created.
  - If path starts with or without '/' and there are several path elements, then a folder will be created with the last path element name under given path minus last path element. The sub-path given should exist, otherwise an error status is returned.

  If openItem is successful, then the new or existing item becomes the current working directory when the item is a folder or a bag.

- int closeItem (itemId idVal)

  Closes the item whose item identifier is idVal value. It returns a status indicating success or failure.

- int removeItem (itemId idVal)

  Removes (i.e. deletes) from the bag the item whose item identifier is indicated by idVal value. It returns a status indicating success or failure. If idVal points to a bag or folder, then the bag or folder must be empty to be deleted.

- int grabItem (itemId idVal, byte *bufPtr, int numBytes, int offset)

Grabs (i.e. reads) numBytes bytes from the item associated with open item identifier idVal into the local buffer pointed to by bufPtr starting at item byte position given by offset. It returns the number of bytes grabbed or an error status indicating cause of failure. If the item identified by idVal is a folder or a bag, then grabItem will grab folder or bag contents respectively and ignore rest of parameters. The bufPtr will have a pointer to an itemId [] table and numBytes will have number of items inside the folder or bag.

- int putItem (itemId idVal, byte *bufPtr, int numBytes, int offset)

  Puts (i.e. writes) numBytes bytes from the local buffer pointed to by bufPtr to the item associated with open item identifier idVal starting at byte position given by offset. It returns the number of bytes put or an error status indicating cause of failure. If the item identified by idVal is a folder or a bag, then putItem will put on persistent storage the current folder metadata or bag metadata contents respectively and ignore rest of parameters.

- int changeItem (char *path)

  Changes current working item (i.e. directory) to path given. It returns a status indicating success or failure. If path does not start with '/', then a relative path to current working directory is assumed.

- int addItemAtt (itemId idVal, char *attName, char *attValue)

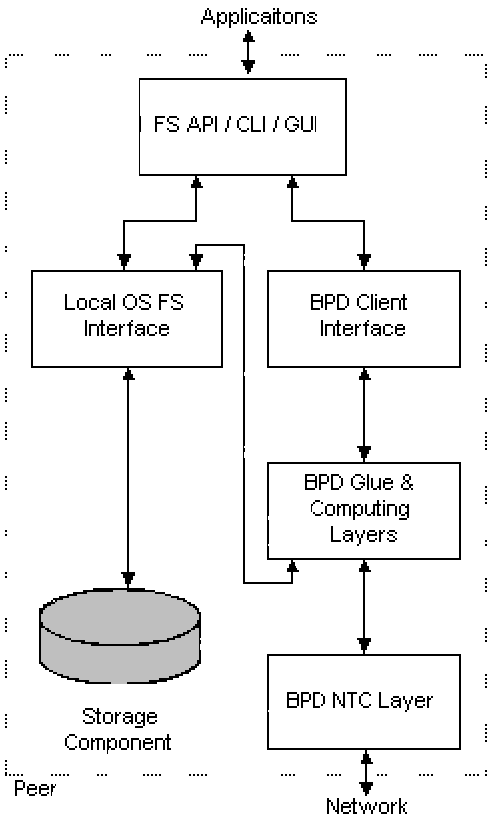
  Adds attribute passed as parameter in attName with value stored in attValue to item identified by idVal. If the attribute exists, then its value is replaced with attValue. It returns a success or failure status.

- int removeItemAtt (itemId idVal, char *attName)

  Removes attribute passed as parameter in attName from the item identified by idVal. It returns a status indicating success or failure.

- ItemId [] searchItem (logOp logVal, char *attName, char *attValue...)

  Search for items that have attributes with their respective values passed as parameters joined by the logical operator specified by logVal. LogVal can be an 'and' or an 'or' operator. It returns a listing with item identifiers that satisfy attributes with values passed as parameters. If an error occurs, then a failure status is returned instead.



BPD API and GUI Architecture
Figure 2

Besides providing a FS API, BPD also provides a Common Line Interface (CLI) and a Graphical User Interface (GUI) to access the local FS and/or the BPD. These multiple client FS interfaces allow a transparent access to both environments. Figure 2 depicts such architecture. The BPD GUI, similar to regular file explorers (e.g. Microsoft File Explorer), allows file storage, retrieval, deletion and search, bag and directory hierarchy management, metadata view, and other FS related operations. The GUI implements transparent and reliable data movements between both environments. The BPD CLI provides similar functionality like the GUI through the use of line commands.

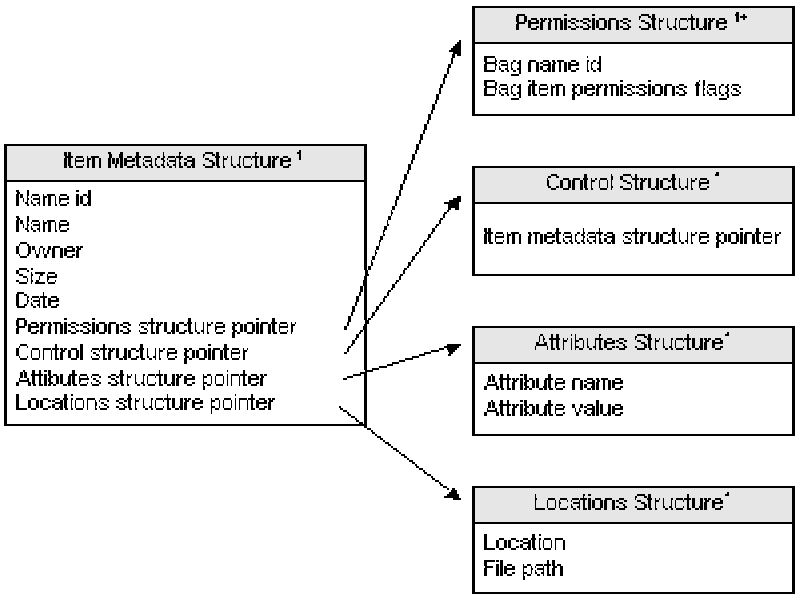
4.2 Data Glue Layer

The Data Glue (DG) layer provides a transparent access to data. It is formed by three components:

- Front-end Data Manager (FDM) responsible to process FS requests and provide handshake to carry on requested operation
- Local-end Data Manager (LDM) controls metadata structures for localized data (i.e. bags) residing somewhere in the P2P system.
- Back-end Data Manager (BDM) controls metadata structures being formed primarily when new bags are built. Once a bag is stable, BDM transfer bag metadata control to LDM.

The DG layer also implements four essential protocols to implement BPD APIs. These four basic algorithms, which are described later in the *Protocols* section, include algorithms for data storage, retrieval, replication and discovery.

There is one basic structure and four auxiliary structures to support metadata associated with *bag items*. A *bag item* (or *item* for simplicity) can be a file, a folder within a bag or the bag itself. Figure 3 shows these structures and the information the structures control.



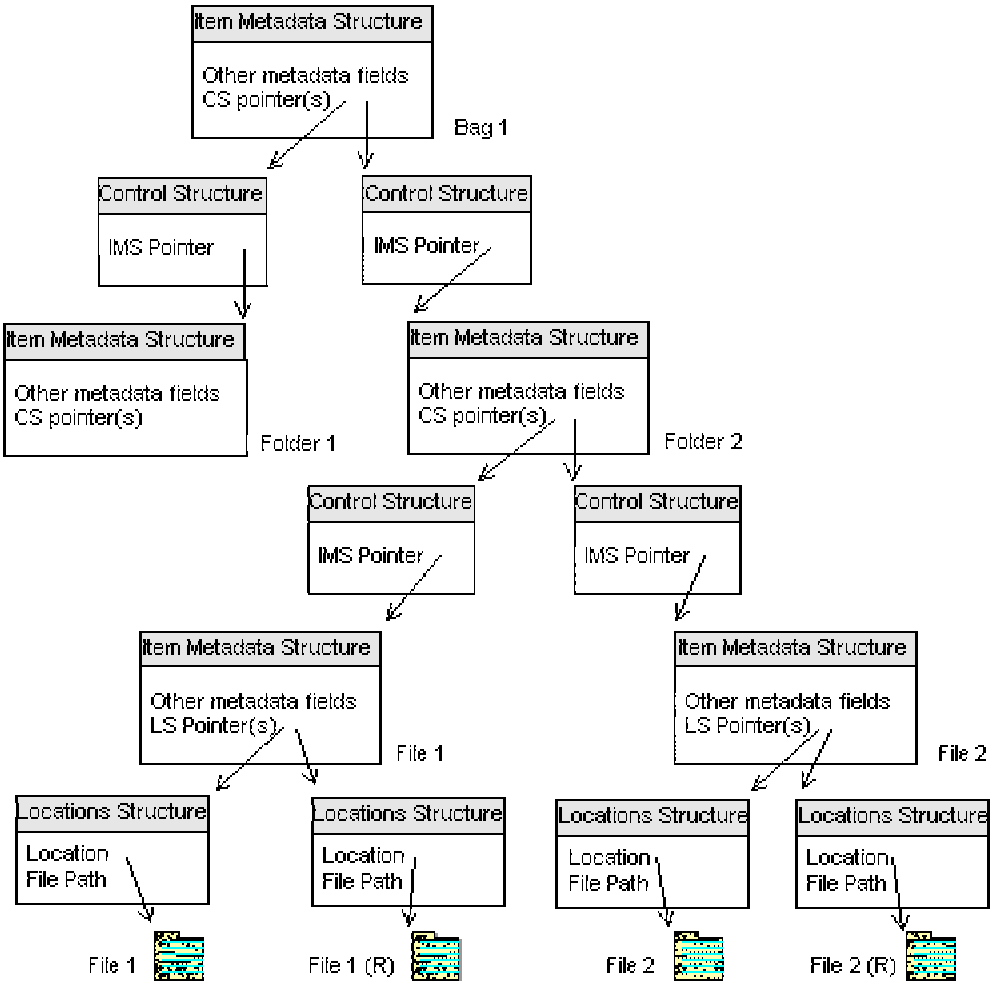
BPD Basic Metadata Structures[1]
Figure 3

The item metadata structure keeps general metadata for a *bag item*. It has item fields for name id, name, owner, size, date (creation, modification, access) and pointers to auxiliary structures.

The bag permissions structure keeps the read/write/execute permission rights for an item within a given bag and the bag name id where the item belongs. If the item is a file, then the item can belong to 1 or more bags according to application dependencies and relationships. There can be different item permissions for each bag where this item

[1] Cardinality per metadata structure: '1' = one, '1+' = at least one or more, '*' = zero or more.

belongs. In the case the item is a bag, the bag name id in the permissions structure will be the same as the name id in the item metadata structure.



BPD Bag Hierarchy Example
Figure 4

Pointers to hold the hierarchical structure of a bag are kept by the control structure. The control structure links an item metadata structure representing a parent item to another item metadata structure representing a child item and so on. An example of this hierarchy linkage is shown in figure 4. This example shows one bag item (i.e. bag 'Bag 1') with 2

bag items (i.e. ‘Folder 1’ and ‘Folder 2’). ‘Folder 1’ is empty and ‘Folder 2’ has 2 bag items in it (i.e. ‘File 1’ and ‘File 2’). The files have a replication factor of 1.

Clients can allocate attributes and its associated value attributes to each bag item. These bag item attributes keep semantics specific to the item. They are used to create, search and access semantically related bags. For example you can search for a bag with attribute name ‘patient name’ that has associated value ‘John Smith’. Once the bag is found, you can access all related bag items to this bag in a hierarchical form. Since the bag is kept locally near, its retrieval is fast and efficient. Consequently, both existing data dependencies and existing data relationships are supported.

Actual physical file location with its associated path is kept in the location structure. If a file is replicated, then several location structures exist (see Figure 4). The BPD has a file level granularity. The file is physically stored on any peer that participates in the web of computing devices, however the computing layer takes care to put bag items that are related locally near. Also, it is possible to extend the location structure to support multiple location pointers; therefore a more detail granularity can easily be achieved (e.g. block level granularity).

When a file is loaded to a local peer from a remote location, a temporal location structure is created to hold file copy pointers. An updated file is written back to original location(s) when the file is closed and the temporal location structure is deleted (see Figure 6).

Auxiliary structures (i.e. permissions, control, attributes and location structures) even though are shown as different entities, they can be implemented as dynamic vectors inside the item metadata structure object (e.g. Java hash vectors). This implementation gives flexibility in the number of pointers and a fast retrieval for the pointers inside the item metadata structure.

The LDM is responsible to maintain (create, store and retrieve) bags' hierarchies created in the local peer. These bags are stored in the peer persistent storage as objects. The bags are dynamically loaded into memory when the BPD is executed at the client peer. When a bag item (i.e. a file) is retrieved or stored, the LDM will contact directly the peer responsible for the item via the computing and NTC layers. The computing layer helps with allocation and retrieval CAS algorithms to efficiently balance resources. The NTC layer helps with the physical communication between peers.

FDM will either assemble or disassemble file blocks to the client application for a write or read operation respectively. When a request comes from the client, FDM immediately knows if it can fulfill the request by checking with LDM for local controlled data, otherwise it asks BDM to start building the bag item that satisfies the request. FDM keeps the working files in local temporal storage as possible to minimize network traffic and minimize access latency. Keeping files in temporal storage also supports the data life cycle of continuous updates for a relatively short period of time before the client deletes

or stores the file (see *Data Mobility* section). Once the file is stored, the LDM keeps control of its associated metadata after the BDM allocated the file somewhere in the distributed system.

When new bag items (i.e. files) are stored, the BDM uses computing layer CAS models to allocate storage within the DFS. The BDM relies on the computing layer to accurately balance resource usage and achieve data locality. Once the bag items are allocated, the BDM passes control of the item metadata to the LDM. Furthermore, the BDM cooperates with other peer BDMs to search for bag items with specifics attributes and form new hierarchies. This cooperation between BDMs is also based on CAS models provided in the computing layer.

4.3 Computing Layer

The *Computing Layer* (CM) provides Complex Adaptive System (CAS) services to the upper layers. The CM layer provides CAS services to allocate, retrieve, and search storage resources across the P2P system. These CAS services are based on simple biological behaviors commonly observed in animal, insect and other societies [20][21]. The CM layer uses the services of the NTC layer to establish linkages to other peers within the BPD system. Thesis chapter 4 "*CAS Algorithms*" describes in detail CAS services and algorithms provided by the CM layer. This section outlines what it is provided by the computing layer and how it interfaces to the DG layer.

* Allocation services are called when the DG layer executes data storage and/or replication protocols (see protocols section). These services find a suitable peer where the file can be assigned. The suitability is achieved by both balancing global distributed system resources and by selecting a peer locally near to its bag contents. The allocation service syntax is:

peerList allocateItem (itemId idVal, int replicationFactor)

If allocateItem is successful, then a peer list of suitable locations is returned where the DG layer can put the item, identified by idVal, for permanent storage. If allocateItem fails then an error status is returned. The replication factor for the item is passed as parameter together with the identifier for the item to be allocated in the DS.

* Retrieval services provide a listing of peers that can provide a copy of the requested item identified by idVal to the DG layer. Retrieval services dynamically selects the best location(s) in descending order of retrieval quality (e.g. latency) that can provide the requested item. The retrieval service syntax is:

peerList retrieveItem (itemId idVal)

If retrieveItem is successful, then a peer list that can provide the item object is returned to the DG layer. Retrieval services are called when the data glue layer executes data access and retrieval protocols. The DG layer will then be responsible to contact directly the selected peer and provide item contents to the application according to permissions and access flags. If retrieveItem fails, then an error status is returned.

* Search services looks up for peers that can provide items with associated attribute-value, which are passed as parameters. This parameter list has a pair for each attribute-value to search. Logical operator logVal indicates if a logical 'and' or a logical 'or' search will be applied on the attribute-value list. The search service syntax is:

peerList searchItem (logOpe logVal, char *attName, char *attValue,...)

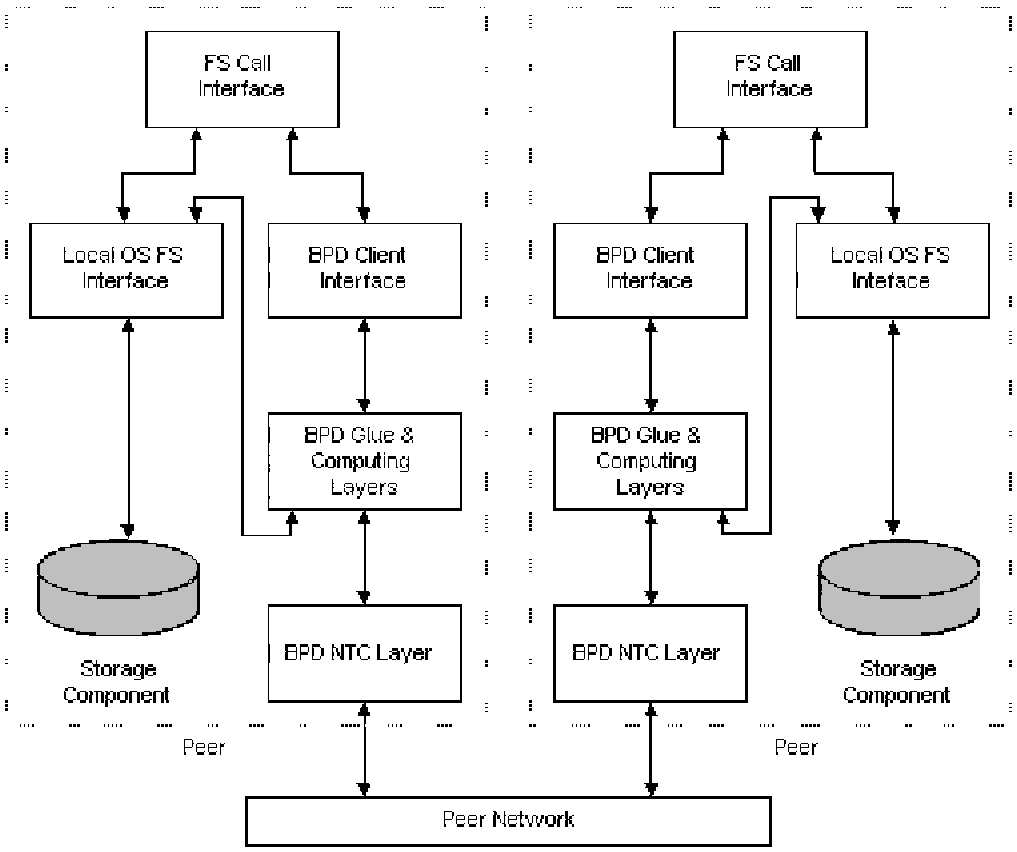
If searchItem is successful, then a peer listing that satisfies the attribute-value list according to the logical operator logVal is returned. Search services are called when searching for bag items and/or constructing bags according to some semantic meaning. The DG layer will be responsible to create the bag and later retrieve the item that the application needs. If searchItem fails, then an error status is returned.

4.4 NTC Layer

At the bottom of the BPD architecture stack, shown in Figure 1, is the network, transport, and communication (NTC) layer that provides services to physically interconnect nodes in the P2P distributed system. The NTC layer relies on existing networking and communication protocols and technologies to connect to other peers.

There are different ways the NTC layer can be implemented. The implementation depends on the way the peers are physically layout within the organization where the BPD is placed. If all peers belong to a well-segmented subnet, then a simple communication protocol among peers is required. A protocol like TCP/IP can be used to connect to other peers in the subnet. The subnet provides the network peers that

participate in the P2P DFS. All communication is socket based between the peers to provide FS services to the client applications.



BPD Peer Network
Figure 5

If a more complex peer network exists in the organization then a more robust NTC layer is required. For example, this scenario occurs when not all the peers in a subnet participate in the BPD, or there are different sites with firewalls and peers belonging to different subnets, *etc*. A P2P platform like JXTA [22], .NET [23], *etc*. is used to implement required services. Platform services of membership, transport, security and lookup among others are required for these physical complex peer networks. All

communication is P2P platform based between the peers to provide FS services to the client applications.

Both DG and computing layers make use of NTC layer services to communicate to other peers. The computing layer executes CAS algorithms that communicate to other peers to work collaboratively and eventually provide emergent global solutions, whereas the DG layer uses a direct communication to other peers to store and retrieve bag items.

Each BPD peer is independent from each other. A peer collaborates to other peers as shown in Figure 5. The NTC layer provides a communication path between them. The peers with their software entities work together when executing CAS models. A global FS outcome emerges that balance resources, maintain locality and scale according to the peer network characteristics, while application domain semantics are supported.

5. Algorithms and Data Models

5.1 Protocols

The basic protocols support essential API function calls available to client applications (e.g. grabItem, putItem, searchItem, *etc*). These protocols run at each peer in the distributed system within the DG layer.

There are four basic BPD architecture protocols supporting the P2P DFS API system calls. These protocols include:

1. Data storage protocol.
2. Data retrieval protocol.
3. Data replication protocol.
4. Data discovery protocol.

The front-end data manager (FDM) implements the API function calls that make use of these protocols. The API function calls are described in the *API Interface* section.

The data storage protocol implements receiving *data chunks*[2] from an API function call process. Then the protocol stores the data chunk in a place somewhere in the distributed system. The pseudo-code for this protocol is:

Data Storage Protocol

1. Data chunk D_i is available to be stored from an API function call process.
2. The FDM pushes D_i metadata towards the local-end data manager (LDM) and the back-end data manager (BDM)
3. The BDM calls the computing layer allocateItem() function.
4. The CAS models implemented in the allocateItem() function find suitable location(s) where D_i can be allocated.
5. The BDM selects location(s) according to a metric (e.g. latency, locality, *etc.*) and updates D_i metadata. BDM communicates this information to the LDM.
6. The BDM marks pending confirmation of storage at selected destination(s).
7. The LDM pushes data chunk D_i to selected destination(s) using NTC layer services.
8. The LDM waits for confirmation from selected destination(s).

[2] We use the term *data chunk* to represent the data granularity managed by a BPD implementation. A *data chunk* can be a data block, a group of data blocks, or a complete file.

9. Once D_i is confirmed stored from selected destination(s), the LDM notifies the BDM. The BDM updates D_i metadata. After the successful or failed operation, the BDM notifies the FDM.
10. The FDM receives success or failure status. Then the FDM notifies the client application of the status.

♦

The data retrieval protocol implements retrieving data chunks for an API function call process. This data chunk comes from a peer(s) located somewhere in the distributed system. A metric (e.g. latency, locality, *etc.*) is used to select the peer(s) providing the data chunk. The pseudo-code for this protocol is:

Data Retrieval Protocol

1. API function call process wants to access a data chunk D_i.
2. The front-end data manager (FDM) asks the local-end data manager (LDM) about D_i.
3. The LDM checks its metadata structures entries for D_i. If the LDM finds D_i in the metadata structures then the LDM executes the computing layer retrieveItem() function. If LDM fails to find Di in the metadata structures, then LDM notifies failure to the FDM.
4. The CAS models implemented in the retrieveItem() function find suitable location(s) where D_i can be retrieved.
5. The LDM selects location(s) according to a metric (e.g. latency, locality, *etc.*).
6. The LDM pulls data chunk D_i from selected source location(s) using NTC layer services.
7. The LDM waits for data chunk D_i to arrive.
8. Once Di is confirmed that has been received correctly from source location(s), the LDM notifies the FDM success or failure status.
9. The FDM receives success or failure. Then the FDM notifies the client application of the status.

♦

The data replication protocol implements a replication scheme for data chunks D_i coming from an API function call process. The basic idea of supporting some level of replication

is for reliability reasons. It is well understood that by having multiples copies of data, system resilience to failures is increased by greater data availability. However, the overhead to synchronize copies in the system decreases performance and capacity. The basic outline for this protocol is:

Data Replication Protocol

1. API function call process wants to replicate a data chunk D_i according to a given replication value.
2. The FDM executes steps a to c for N times. N is the replication factor.
    a. If is this a new replica copy? Then the FDM creates a new D_i location structure metadata and updates item metadata structure pointers
    b. If is this an update to an existing replica copy? Then the FDM selects one of the D_i location structure metadata
    c. The FDM executes the data storage protocol with selected D_i metadata
3. Once the FDM receives success or failure for replicated copies, the FDM notifies the client application of the replication status.

♦

If a selected site fails to replicate a data chunk D_i, then the FDM only registers those sites where D_i was actually stored. In the case of a site failing to replicate D_i, the FDM proactively initiates a process to recover those replica copies at other sites. This keeps D_i replicated as requested by the API function call process.

Data discovery protocol implements search for a bag requested by an API function call process. This bag has associated attribute-value pairs and it may or may not exist in the local structures. In the first version being implemented, the BPD only supports discovery at the bag level. Other versions will allow searching for bag items as well. The basic layout of this protocol is:

Data Discovery Protocol

1. The FDM receives a request from an API function call process to search a bag with associated attribute-value pairs.
2. The FDM looks up in the metadata structures for this bag.
3. If the bag is found in local metadata structures, The FDM creates a partial response with associated bag metadata. Otherwise, the FDM creates an empty partial response.
4. The FDM asks the BDM to collaborate with other locations' BDMs to find more bags with associated attribute-value pairs.
5. The BDM calls the computing layer searchItem() function.
6. The CAS models implemented in searchItem() function collaborate and work together with other BDMs to find requested bag.
7. Once the BDM starts receiving bag location responses, it will inform the FDM. The FDM will add these responses to the partial response created in step 3.
8. After a timeout, the FDM responds to the API process with partial bag results.
9. The FDM asks API process if it continues with more searches for the same bag. If the API process does not want to continue, then the FDM cancels searching for this bag and returns to the API process, otherwise it continues receiving bag results.
10. The FDM goes back to step 7 to continue for more bag responses.

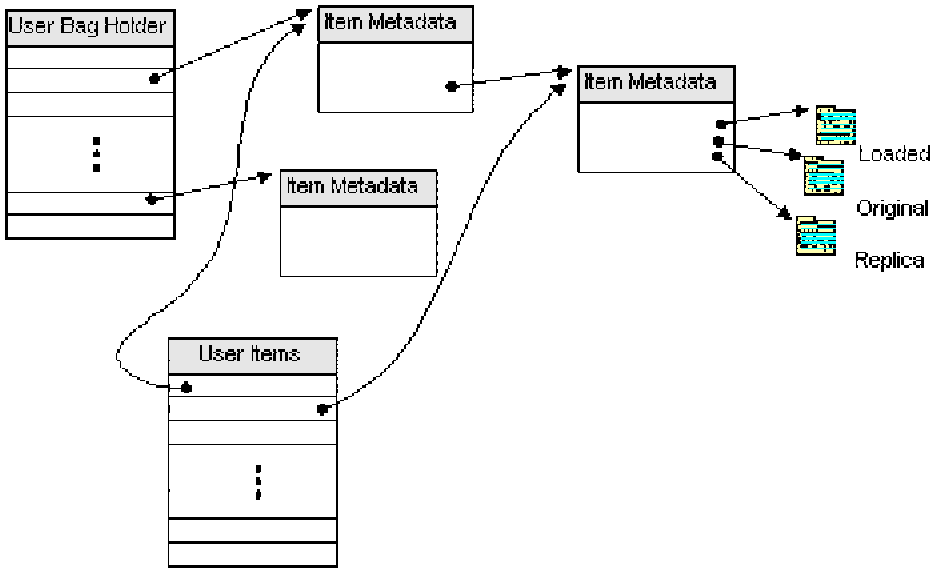
◆

5.2 User Control Tables

There are two user control tables to keep pointers to loaded bag hierarchies and opened user items. The *user bag holder* table represents the bags' holder introduced in *BPD Architecture Overview* section. It has an entry for each bag hierarchy loaded into local memory and available for FS operations. The *user items* table has an entry for each bag item identifier that it is opened and directly accessible. Figure 6 shows an example of the relationship in the user control tables.

In Figure 6 example, there are two bags loaded in the user client session. One bag has a bag item (i.e. a file item metadata with three pointers to the original, replicated and

loaded file copies) and the other bag is empty. The bag holder table keeps pointers to both bags. Furthermore, the user items table keeps direct pointers to opened bag items, in this case a pointer to a bag and a pointer to a file.



User Control Tables
Figure 6

5.3 Data Mobility

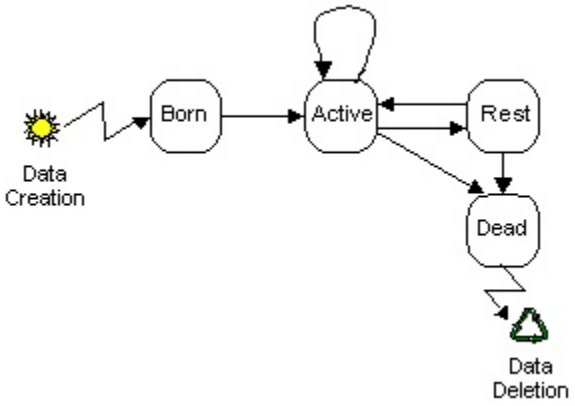
File data lives a continuum of states, from the time it is created until finally it is deleted. Figure 7 shows the data life cycle supported by the BPD.

These states represent the various states file data exists in within the P2P DFS.

- Born state – data has been created and resources need to be allocated to store it. The DG layer allocates item metadata structures.
- Active state – data is being accessed and used by applications. It is allocated locally for online access (e.g. local cache, main memory, local disk). Data grows

and shrinks per application activity. When data is closed, it transitions to rest state for its allocation in the P2P system, the CM layer allocates space for the data using CAS algorithms, and the DG layer updates item metadata structures as per application activity carried on the data.

- Rest state – data resides on storage resources for subsequent access in the future. When data is opened for access, it transitions to active state to be used by applications, the CM layer searches and retrieves best locations for the DG layer, and then the DG layer retrieves the data updating corresponding metadata structures. When the data that is in rest state has not been accessed for a certain period of time (e.g. lease expiration); it may be either backup, compressed or moved in to the dead state for deletion.
- Dead state – data has been trashed and its previously allocated resources can be recycled. Its metadata structures are deleted.



Data Life Cycle
Figure 7

5.4 Data Consistency

Semantics for file sharing are relaxed. The BPD implements a combination of session semantics with immutable files. Session semantics are supported from the perspective that modifications to an open file are only visible locally to the client that makes the updates to the file. After the file is closed and stored in the distributed system, then those modifications will be available to the other clients. Clients that have cached previously file copies have old data and must re-read the file to update its contents.

The storage virtual layer that it is formed by all BPD peers only supports create and read operations on it. When a file is locally updated, the new copy overwrites an existing one when it is allocated within the distributed system with the same name. Even though files cannot be updated in the distributed system, the metadata that controls them can be. Consequently, from this perspective the virtual layer supports immutable files that can be updated locally. If a file is read while it is replaced, BPD detects that the file has changed and notifies those clients reading from the file.

6. Current and Future Work

We are currently implementing BPD. Complex adaptive system (CAS) algorithms for storage resource allocation have been designed, developed and tested [20]. The CAS models have allocated resources efficiently in a simulated P2P DFS of several thousands peers. CAS models have been tested on P2P systems of up to more than 25,000 peers. Furthermore, results reliability achieved indicates that the CAS models scalability can go

much larger than this P2P size value. The initial work is being expanded to develop CAS algorithms for data retrieval and search. All together the CAS algorithms will provide the computing layer mechanisms that dynamically adapt, scale and efficiently utilize distributed resources. The CAS algorithms with their implicit locality foster the data spatial and temporal locality that it is required in the bag hierarchies. Then mechanisms that implement the API function calls will be developed to provide FS services to applications. This API function calls will make use of CAS algorithms to work within the distributed system. The emergent biological CAS properties will provide global solutions for data storage and retrieval for the P2P DFS.

7. Conclusion

The Biologically-Inspired Peer-to-Peer Distributed File System (BPD) is expected to provide file system services in an environment that it is dynamic, self-organized, ad-hoc and decentralized with application semantic support. Some of the characteristics that make BPD distinctive include:

CAS models for storage, retrieval and search Complex adaptive systems models provide emergent global results, which are the result of simple, individual and quasi-independent behavior of community members. The swarm intelligence that emerges from the global behaviors of simple members boasts autonomy and self-sufficiency, which allows them to adapt quickly to changing environmental conditions. CAS present team or

group models that solve complex distributed problems, which otherwise will be difficult with deterministic centralized or distributed techniques.

Data hierarchies and relationships Bags hierarchies maintain and support application dependencies and their respective proximity within the hierarchy. The concept of a bag is introduced to contain related data within a logical boundary that has some meaning and it is useful to the application within the same context. Many bags can coexist to represent independent related information that it is easily accessible.

Commodity peers for storage and retrieval Even that its common in today's P2P DFS, each peer contributes and use resources according to their capabilities. BPD assumes an ad-hoc behavior for the peers with no central server or controller. CAS models take care of the dynamics of the peers. Allocation algorithms developed [20] have demonstrated that thousands of peers with different capabilities can collaborate and work independently to provide storage balance and good resource utilizations across the entire system.

Common file life cycle File data usually follows a continuous sequence of modifications for a relatively short period of time to be either deleted or stored. Once the file is stored, it receives few modifications, however several read retrievals occur before it is either deleted or backed up. The BPD supports this common file life cycle to manage data more efficiently among the peers. This data cycle brings better re-use of existing resources and avoids its waste. This is achieved by locally processing initial transient access and once

the file data is stable the BPD leverages the distributed resources by allocating the data into the P2P system.

Acknowledgements

We want to thank Peter Graham, John Anderson and Jeff Diamond from University of Manitoba for their participation in useful discussions on BPD. Also thank the St. Boniface Research Centre and the Medical Informatics Group by sponsoring this research work.

References

[1] Milojicic D, Kalogeraki V, Lukose R et al. "*Peer-to-Peer Computing*", Hewlett Packard Technical Report HPL-2002-57, HP Laboratories Palo Alto, CA USA, pp 1-51, March 2002.

[2] Bonabeau E. and Theraulaz G. "Swarm Smarts", Scientific American, Vol. 282, No. 3, pp 54-61, March 2000.

[3] Ghemawat S, Gobioff H and Leung S, "The Google File System", Proceedings of the 19th ACM Symposium on Operating System Principles, Bolton Landing, NY USA, October 2003.

[4] Cort R. "*Understanding Windows NT POSIX Compatibility*", Microsoft Corporate Technology Team Technical Report, USA, pp 1-7, June 1993.

[5] The IEEE and The Open Group, "*IEEE Std 1003.1, 2003 Edition*", The Open Group Base Specification Issue 6, 2003 Edition, 2003.
Available at http://www.unix-systems.org/single_unix_specification/

[6] Dabek, F., Kaashoek, F., Karger, D., Morris, R., and Stoica, I., "*Wide-Area Cooperative Storage with CFS*", Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP'01), pp 202-215, Banff Canada, October 2001.

[7] Rowstron, A., and Druschel, P., "*Storage Management and Caching in PAST, a Large-Scale, Persistent Peer-to-Peer Storage Utility*", Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP'01*),* pp 188-201, Banff Canada, October 2001.

[8] Stoica I., Morris R, Karger D, Kaashoek M and Balakrishnan H, "*Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications*", Proceedings of the ACM SIGCOMM 2001 Technical Conference, pp 149-160, San Diego CA, Aug 2001.

[9] Druschel P and Rowstron A, "*Pastry: Scalable, Distributed Object Location and Routing for Large-Scale Peer-to-Peer Systems*", Proceedings of the 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001), pp 329-350, Heidelberg Germany, November 2001.

[10] Keleher P, Bhattacharjee B and Silaghi B, "*Are Virtualized Overlay Networks Too Much of a Good Thing?*", 1st International Workshop on P2P Systems (IPTPS 2002), pp 225-231, MIT Labs, Cambridge MA, March 2002.

[11] Rhea, S., Wells, C., Eaton, P., Geels, D., Zhao, B., Weatherspoon, H., and Kubiatowicz, J., "*Maintenance-Free Global Data Storage*", IEEE Internet Computing, Vol. 5, No. 5, pp 40-49, September-October 2001.

[12] Zhao B, Kubiatowicz J and Joseph A, "*Tapestry: An Infrastructure for Fault-Tolerant Wide-Area Location and Routing*", Technical Report UCB/CSD-01-1141, University of California at Berkeley, Computer Science Dept., April 2001.

[13] Thekkath C, Mann T, Lee E, "*Frangipani: A Scalable Distributed File System*", Proceedings of the 16th ACM Symposium on Operating Systems Principles, pp 224-237, 1997.

[14] Anderson T, Dahlin M, Neefe J, Patterson D, Roselli D and Wang R, "*Serverless Network File Systems*", Proceedings of the 15th ACM Symposium on Operating Systems Principles, Copper Mountain, CO USA, pp 109-126, December 1995.

[15] Ousterhout J, Da Costa H, Harrison D, Kunze J, Kupfer M and Thompson J, "*A Trace-Driven Analysis of the Unix 4.2 BSD File System*", Proceedings of the 10th Symposium on Operating System Principles (SOSP'85), Orcas Island, Washington USA, pp 15-24, December 1985.

[16] Thomson J, Adams D, Cowley P and Walker K. "*Metadata's Role in a Scientific Archive*", IEEE Computer, Vol. 36, Num.12, pp 27-34, December 2003.

[17] Gifford D, Jouvelot P, Sheldon M, and O'Toole J. "*Semantic File Systems*", Proceedings of the 13th ACM Symposium on Operating Systems Principles, SOSP'91, pp 16-25, 1991.

[18] Hansen D, and Adams D. "*A Database Approach to Data Archive Management*", Proceedings of the 1st IEEE Metadata Conference, Silver Spring, Maryland USA, April 1996. Available at www.computer.org/conferences/meta96/hansen/paper.html

[19] Jeffrey K. "*Metadata: An Overview and Some Issues*", Proceedings of the 11th ERCIM Database Research Group Workshop Metadata for Web Databases, Sankt Augustin, Germany, May 1998.

Available at http://www.ercim.org/publication/ws-proceedings/11th-EDRG/jefferey.pdf

[20] Camorlinga S, Barker K, Anderson J. "*Multiagent Systems for Resource Allocation in Peer-to-Peer Systems*", Proceedings of the Winter International Symposium on Information and Communication Technologies WISICT 2004, Cancun Mexico, pp 173-178, January 2004.

[21] Anderson C, Franks N. "*Teams in Animal Societies*", Behavioral Ecology, Vol. 12, No. 5, pp 534-540, September 2000.

[22] The JXTA Project, http://www.jxta.org/

[23] Microsoft .Net Architecture, http://www.microsoft.com/net

[24] Rodeh O and Teperman A, "*zFS – A Scalable Distributed File System using Object Disks*", 20th IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies (MSS'03), San Diego California, pp 207-218, April 2003.

[25] Stein C, Tucker M and Seltzer M, "*Building a Reliable Mutable File System on Peer-to-Peer Storage*", Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems (SRDS 2002), Osaka University, Suita, Japan, pp 324-329, October 2002.