1. Introduction

The figure above illustrates the components of the Shared Storage architecture. A server manages disks and other persistent memory devices. It presents an efficient, low-level model of storage to clients via a well-defined network protocol. The clients use this low-level architecture to implement different, high-level storage systems like databases and file systems.

The protocol with which clients communicate with servers involves the exchange of packets. Each exchange consists of the transmission of a packet from the client followed by a response packet from the server. The network protocol used between the client and server need not be reliable, nor need it be based on connections between the client and server. The definition of the packets used in this exchange is called the Shared Storage Wire API or SSWAPI.

The way in which storage is managed on the disk in order to present the shared storage abstraction to the client is specified by the Shared Storage Disk API or SSDAPI. This mechanism is specified so servers can be written in Java and C/C++ can be used interchangeably. (In the future, efficient storage servers may be implemented with different disk API's.) The SSDAPI should be not relevant to the clients.

This document consists of the three sections. Section 2 discusses storage model presented by the server to the client. The format of the packets on the wire protocol is discussed in Section 3. The disk layout and API is discussion in Section 4. (Section 4 is currently being revised as the server is being implemented. Section 5 is left over from the previous version of the document and will be corrected as soon as the client is functional.)
2. Basic Model

The storage server provides clients with storage. The smallest unit of storage is a byte (8 bits). A storage unit is a collection of up to $2^{63}$ bytes. The total range of possible bytes of a storage unit is called the storage unit’s address space. Specific bytes in a storage unit are indicated by 8-bit signed integers, called addresses. Negative addresses are reserved for special uses by the Shared Storage System.

Storage units are identified by a globally unique Storage Identifier or StorageID. StorageID’s are 8 bytes long, so there can be as many as $2^{64}$ different storage units in the system. StorageID’s are never reused.

Storage units come in two different flavors: data storage units and directory storage units. Data storage units have positive StorageID’s and directory storage units have negative StorageID’s. Data storage units may contain up to $2^{63}$ bytes of data and are the raison d’être of the shared storage system. Directory storage units provide a mapping between strings and StorageID’s. The StorageID –1 always refers to the root directory storage unit of the system.

The data bytes in a storage unit are in one of two states: uninitialized and initialized. Uninitialized bytes have no associated data, and may have no backing memory. Writing data to an address initializes the byte if necessary, and associates data with the address. Once a byte has been initialized, it cannot be returned to the uninitialized state. Reading of an uninitialized byte is illegal and produces an exception.

Clients may create storage units. When created, the address space of a storage unit is empty. Clients may write data to a storage unit, assigning the data to any of the $2^{63}$ byte addresses, in any order. (Data does not need to be written sequentially.)

3. Wire Protocol

All interactions between the client and server consist of a packet send from the client to the server and response packet sent from the server to the client. Each exchange is independent from previous and subsequent exchanges. The server does not need to maintain any information about the client.

To minimize the number of different packet formats and thus simplify the software, the server and client sides often use the same message formats. Thus in response to a READ request from the client, the server sends a WRIT message to the client.

Most messages contain a field for the client’s credentials. Credentials are unforgeable tokens that identify a client and the client’s capabilities. The design of such a token is discussed elsewhere. The credentials 0 are used to indicate the default credentials for any particular task.

The messages used by the storage system are divided into four classes: creation/deletion operations, data transfer operations, name management, and lock management. Each class of messages is discussed in one of the following sections. Section 3.5 discusses the acknowledgement message and acknowledgement codes, which is used by messages of all classes.

---

1 This restriction needs to be removed in the future. This will require adding a new packet to the wire protocol.
The initial implementation will be using UDP. In a networking course, the students could do a TCP implementation. The Java implementation provides a procedural interface to the Wire Protocol.

When strings are mentioned as the size of a field, then they are encoded as a count (32 bits) of the number of Unicode characters in the string followed by the Unicode characters themselves. Thus a string of 5 characters will require 4 bytes (for the count) + 10 bytes (2 bytes per character) or 14 bytes all together.

The following sections discuss each class of messages.

### 3.1 Creating and Deleting Storage Units

To create a storage unit, the client sends a “Create Storage” command to the server. This command indicates how much data space to initially allocate for the storage unit. If the requested size is positive, then the storage unit is a data storage unit and that many bytes of storage are initially reserved for the new storage unit. If the size is negative then, then a directory storage unit is created. The size parameter indicates that space should be reserved for size directory entries.

The response to a successful “Create Storage” command is also a “Create Storage” command, but with the various fields filled in. To create a storage unit the client uses an exchange like the following.

```
<CREA 0 0 1000000>  ⇒  Request a million bytes of memory
⇐<CREA 0 163 100000>  Only got 100000 bytes, ID of 163
```

The contents of a message is surrounded by angle brackets. In these two messages, the first 4 bytes are an integer that represents the four ASCII characters “CREA”. The type of packet being sent is indicated by these first four bytes. These are then followed by three fields, separated by spaces. In this case each of the three fields is an integer. In general, the fields can be bytes, integers or strings. The precise specification of the details of the packet format, for each type of packet, is given later in this section.

To create a directory storage unit, a Create Directory message is sent instead. See Section 3.3 for further details.

Note that although the client requested a storage unit with space for $10^6$ bytes of data, the server was only able to provide a storage unit with $10^5$ bytes of data. The client has two choices. It can accept this storage unit and hope that when it needs to write more than $10^5$ bytes of data into the storage unit, additional space will be available. Alternatively, it can reject the storage unit provided, delete it, and try to acquire one from a different shared storage server.

To delete the storage unit, the client initiates an exchange like

```
<DELE 0 163>  ⇒  Delete storage unit 163
⇐<ACKN 0>  OK (success)
```

The one field in the acknowledgement packet is the return code. Zero indicates successful completion of the request but provides no additional information. A complete listing of acknowledgement codes is given in Section 3.5. The StorageID of a deleted storage unit may not be reused. It is dead forever (in the current design).

The deleted storage unit has not yet been used, no data has been stored in the data storage unit, and no references to it have been created. Deleting such a storage unit cannot lead to semantic inconsistencies. However, once references to a storage unit have been stored in a directory or propagated throughout the network, one must be more
careful. In particular, inconsistencies in the directory structure can arise, and storage leaks can be created unless garbage collection or a similar mechanism is invoked. Some attention needs to be paid to this.

The following tables detail the format of the Create Storage and Delete Storage packets.

### Create Storage

<table>
<thead>
<tr>
<th>Field</th>
<th>Size</th>
<th>Meaning/contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command</td>
<td>4 bytes</td>
<td>CREA = 0x4352 4541</td>
</tr>
<tr>
<td>Credentials</td>
<td>8 bytes</td>
<td>Client's Credentials</td>
</tr>
<tr>
<td>StorageID</td>
<td>8 bytes</td>
<td>StorageID created (or zero) when requesting storage be created.</td>
</tr>
<tr>
<td>Size</td>
<td>8 bytes</td>
<td>Initial size of storage to reserve (but not necessary allocate), or how much was actually allocated.</td>
</tr>
</tbody>
</table>

### Delete Storage

<table>
<thead>
<tr>
<th>Field</th>
<th>Size</th>
<th>Meaning/contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command</td>
<td>4 bytes</td>
<td>DELE = 0x4445 4C45</td>
</tr>
<tr>
<td>Credentials</td>
<td>8 bytes</td>
<td>Client's Credentials</td>
</tr>
<tr>
<td>StorageID</td>
<td>8 bytes</td>
<td>StorageID to be deleted.</td>
</tr>
</tbody>
</table>

### 3.2 Accessing Data

Data is accessed via the Read Data and Write Data packets.

To read data from a data storage unit, clients must specify the storage unit to be read (via a StorageID), where to start reading, and how many bytes are desired. If successful, the desired data is returned via a Write Data packet (the server is “writing data” to the client). Unsuccessful requests (uninitialized data, nonexistent storage unit, not a data storage unit, etc) are answered with a negative acknowledgement packet. On successful Reads, the server returns a contiguous subset of the data requested that begins with the first byte requested. If the number of bytes to be read is negative, then all the bytes until the next invalid byte are being requested.

To write data to a data storage unit, the client sends a Write Data packet is server. The server then responds with an acknowledgment packet for both successful and unsuccessful writes. The storage server guarantees that on unsuccessful writes, the storage unit is left undisturbed, while on successful writes the entire data provided by the client is written to the storage unit.

The format of the Read and Write packets is given in the following tables. If the length field is negative, then indicates that all the contiguous, initialized data starting at the indicated byte address is being requested.

### Read Data

<table>
<thead>
<tr>
<th>Field</th>
<th>Size</th>
<th>Meaning/contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command</td>
<td>4 bytes</td>
<td>READ = 0x5245 4144</td>
</tr>
<tr>
<td>Field</td>
<td>Size</td>
<td>Meaning/contents</td>
</tr>
<tr>
<td>---------------</td>
<td>------</td>
<td>------------------------------------------------------</td>
</tr>
<tr>
<td>Command</td>
<td>4 bytes</td>
<td>WRIT = 0x5752 4954</td>
</tr>
<tr>
<td>Credentials</td>
<td>8 bytes</td>
<td>Client’s Credentials or 0</td>
</tr>
<tr>
<td>StorageID</td>
<td>8 bytes</td>
<td>StorageID to be written</td>
</tr>
<tr>
<td>AttributeID</td>
<td>4 bytes</td>
<td>Which attribute of the storage unit to modify</td>
</tr>
<tr>
<td>Offset</td>
<td>8 bytes</td>
<td>First byte to write</td>
</tr>
<tr>
<td>Length</td>
<td>8 bytes</td>
<td>Number (n) of bytes of data</td>
</tr>
<tr>
<td>Data</td>
<td>(n) bytes</td>
<td>Data to be written.</td>
</tr>
</tbody>
</table>

Storage units have other attributes other than the data they contain. Among these attributes are their creation date, creator, date of last access, credentials required to access it, etc. Attributes are identified by an **AttributeID**. AttributeID = 0 identifies the data portion of a data segment. The other, well know, attributes are identified in the following table.

<table>
<thead>
<tr>
<th>AttributeID</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Data portion of a data storage unit</td>
</tr>
<tr>
<td>1</td>
<td>Creation date in ?? format</td>
</tr>
<tr>
<td>2</td>
<td>Creator</td>
</tr>
</tbody>
</table>

The following exchange illustrates the packets used to write an 11-character, Unicode string to the data storage unit with StorageID 163. Remember that Unicode uses 16-bits for each character.

\[
\text{<WRDA 0 163 1023 22 "Hello World"> ⇒ Write a string, starting at byte 1023} \\
\text{⇐ <ACK 0 > Got it} \\
\]

Reading two words from the storage unit might produce the following exchange of packets

\[
\text{<REDA 0 163 0 1024 8> ⇒ Read two words (8 bytes) of data} \\
\text{⇐ <WRDA 0 163 0 1023 4 0x65006C00> Here is one word of data} \\
\]

\[^2\] The initial implementation does not implement attributes. Thus, it is not possible to determine the size of a file, creator, creation date, etc. In a real implementation, the different attributes would be represented symmetrically. Furthermore, the Attribute field is missing in the current implementation of the wire protocol. This should be fixed.
In this case, the client will need to perform an additional request in order to retrieve the second word of data. If the client wanted to read the rest of the storage unit starting at byte 1024, the following packet would be sent.

<REDA 0 163 0 1024 - 1> ⇒ Read lots of data

### 3.3 Directory Storage Units

The directory storage units establish a mapping between strings and StorageID’s. Directory storage units are created using the **Create Directory** packet. When requesting a directory storage unit be created, a storage ID field of zero is used. The server will then respond with another **Create Directory** packet, but will indicate in the Storage ID field the StorageID of the directory storage unit created.

<table>
<thead>
<tr>
<th>Create Directory</th>
<th>Field</th>
<th>Size</th>
<th>Meaning / contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command</td>
<td>4 bytes</td>
<td>CRDR = 0x4352 4452</td>
<td></td>
</tr>
<tr>
<td>Credentials</td>
<td>8 bytes</td>
<td>Client’s Credentials</td>
<td></td>
</tr>
<tr>
<td>StorageID</td>
<td>8 bytes</td>
<td>StorageID created (or zero) when requesting storage be created.</td>
<td></td>
</tr>
</tbody>
</table>

The **DeleteStorage** packet can be used to delete a Directory Storage Unit.

The internal structure of the directory storage units is hidden from the client, for both performance and robustness reasons. The client can manipulate directory storage units using the packets described in this section.

The strings used to name a storage unit are strings of up to $2^{32} - 1$ Unicode characters. This allows the client to implement hierarchical file systems using either a single large directory storage unit using absolute file names with embedded delimiters, or with one directory storage unit per directory in the hierarchical file system.

A string can be looked up in a directory using the **Lookup** packet. The result will be returned in a **Bind** packet. The following exchange might occur when trying to lookup the Unicode name “Euclid p and ??? .txt”, and discovering that in the directory storage unit 163, it is associated with StorageID 1093.

<LOOK 0 163 20 “Euclid p and ??? .txt”> ⇒ Lookup the name “Euclid p and ??? .txt”
⇐ <BND 0 163 109320 “Euclid p and ??? .txt”> In 163, it corresponds to 1093

The client creates an association between a storage unit and a name using the **Bind** packet. To associate “Euclid p and ??? .txt” with StorageID 1093 in directory storage unit 163 would yield the following exchange.

<BND 0 163 109320 “Euclid p and ??? .txt”> ⇒ Bind this name to 1093
⇐ <ACKN 0> No problem, done

The storage server checks that the StorageID provided actually refers to a valid storage unit and returns an error code if it does not. A valid storage

---

3 Note that there are Greek and Hebrew characters in this name.
To delete the binding from the directory in storage unit 163, one would do the following:

<UBND 0 163 1093 20 "Euclid p and ???.txt"> ⇒ Bind this name to 1093
⇐ <ACKN 0> No problem, done

Storage unit 0 is always a directory storage unit and is created when the Shared Storage Server is initialized.

The following tables give the precise structure of the different packets used to manage directories.

### Lookup Name

<table>
<thead>
<tr>
<th>Field</th>
<th>Size</th>
<th>Meaning/contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command</td>
<td>4 bytes</td>
<td>LOOK = 0x4C4F 4F4B</td>
</tr>
<tr>
<td>Credentials</td>
<td>8 bytes</td>
<td>Client’s Credentials</td>
</tr>
<tr>
<td>Directory</td>
<td>8 bytes</td>
<td>StorageID that identifies a directory</td>
</tr>
<tr>
<td>Length</td>
<td>4 bytes</td>
<td>Length (n) of Name field</td>
</tr>
<tr>
<td>Name</td>
<td>2n bytes</td>
<td>Name of the storage unit being looked up, in Unicode characters</td>
</tr>
</tbody>
</table>

### Bind Name

<table>
<thead>
<tr>
<th>Field</th>
<th>Size</th>
<th>Meaning/contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command</td>
<td>4 bytes</td>
<td>BIND = 0x4249 4E44</td>
</tr>
<tr>
<td>Credentials</td>
<td>8 bytes</td>
<td>Client’s Credentials</td>
</tr>
<tr>
<td>Directory</td>
<td>8 bytes</td>
<td>StorageID that identifies a directory</td>
</tr>
<tr>
<td>StorageID</td>
<td>8 bytes</td>
<td>StorageID to be registered</td>
</tr>
<tr>
<td>Length</td>
<td>4 bytes</td>
<td>Length (n) of Name field</td>
</tr>
<tr>
<td>Name</td>
<td>2n bytes</td>
<td>Name under which the StorageID should be registered, in Unicode characters</td>
</tr>
</tbody>
</table>

### Unbind Name

<table>
<thead>
<tr>
<th>Field</th>
<th>Size</th>
<th>Meaning/contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command</td>
<td>4 bytes</td>
<td>UBND = 0x5542 4E44</td>
</tr>
<tr>
<td>Credentials</td>
<td>8 bytes</td>
<td>Client’s Credentials</td>
</tr>
<tr>
<td>Directory</td>
<td>8 bytes</td>
<td>StorageID that identifies a directory</td>
</tr>
<tr>
<td>Length</td>
<td>4 bytes</td>
<td>Length (n) of Name field</td>
</tr>
<tr>
<td>Name</td>
<td>2n bytes</td>
<td>Name that should be removed from the directory.</td>
</tr>
</tbody>
</table>

Note that there is no mechanism to list the contents of a directory storage unit. This is done using the Read packet requesting the entire contents of the storage unit. The format in which data is returned has not yet been defined. Since it may take several reads to receive the entire directory, the directory object may need to be locked to get consistent data.
3.4 Locking Mechanisms

The locking mechanisms are used to implement databases and transactional systems. They are designed so that any granularity of locking needed by an application can be accommodated. Any subset of a storage unit that is allocated can be locked by a single client.

<table>
<thead>
<tr>
<th>Field</th>
<th>Size</th>
<th>Meaning/contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command</td>
<td>4 bytes</td>
<td>LOCK = 0x4C4F 434B</td>
</tr>
<tr>
<td>Credentials</td>
<td>8 bytes</td>
<td>Client's Credentials</td>
</tr>
<tr>
<td>StorageID</td>
<td>8 bytes</td>
<td>StorageID to be locked</td>
</tr>
<tr>
<td>Location</td>
<td>8 bytes</td>
<td>First byte to be locked</td>
</tr>
<tr>
<td>Length</td>
<td>8 bytes</td>
<td>Total number of bytes to be locked</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Field</th>
<th>Size</th>
<th>Meaning/contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command</td>
<td>4 bytes</td>
<td>UNLK = 0x554E 4C4B</td>
</tr>
<tr>
<td>Credentials</td>
<td>8 bytes</td>
<td>Clients Credentials</td>
</tr>
<tr>
<td>StorageID</td>
<td>8 bytes</td>
<td>StorageID to be locked</td>
</tr>
<tr>
<td>Location</td>
<td>8 bytes</td>
<td>First byte to be unlocked</td>
</tr>
<tr>
<td>Length</td>
<td>8 bytes</td>
<td>Total number of bytes to be un-locked</td>
</tr>
</tbody>
</table>

3.5 Acknowledgement

Many operations consist of a single command from the client to the server. The server then needs only to perform the command, no data or information is returned to the client. In this case, the response packet from the server to the client is an acknowledgement packet. The following table lists the condition codes that are returned by the current version of the shared storage system software. At some point, addition conditions need to be assigned and they should be ordered in a more logical manner.

<table>
<thead>
<tr>
<th>Condition</th>
<th>Return Code</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>OK</td>
<td>0</td>
<td>Success, no additional information</td>
</tr>
<tr>
<td>IOError</td>
<td>–1</td>
<td>Some type of I/O error occurred.</td>
</tr>
<tr>
<td>BadWireAPI</td>
<td>–2</td>
<td>Illegal WireAPI message format</td>
</tr>
<tr>
<td>CreateDataFail</td>
<td>–3</td>
<td>Create Data Storage Unit failed</td>
</tr>
<tr>
<td>CreateDirFail</td>
<td>–4</td>
<td>Create Directory Storage Unit failed</td>
</tr>
<tr>
<td>DeleteFail</td>
<td>–5</td>
<td>Delete Storage Unit fail</td>
</tr>
<tr>
<td>ReadDataFail</td>
<td>–6</td>
<td>Reading from a Data Storage Unit failed</td>
</tr>
<tr>
<td>BadBlockAlloc</td>
<td>–7</td>
<td>Blocks badly allocated (overlapping)</td>
</tr>
<tr>
<td>UnitMismatch</td>
<td>–8</td>
<td>Wrong type of Storage Unit while doing a restore</td>
</tr>
<tr>
<td>NotStorageID</td>
<td>–9</td>
<td>Expected a Storage ID and didn’t get it.</td>
</tr>
</tbody>
</table>
The **Acknowledgement** packet is used when returning an acknowledgement. The format of this packet is given below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Size</th>
<th>Meaning/Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command</td>
<td>4 bytes</td>
<td>ACKN = 0x4143 4B4E</td>
</tr>
<tr>
<td>Data</td>
<td>8 bytes</td>
<td>Acknowledgement data. A negative number indicates an error has occurred and provides additional data. Zero or a positive number indicate success.</td>
</tr>
</tbody>
</table>

## 4. The Server

The server manages blocks of storage called **storage units**. A storage unit contains no more than $2^{63}$ bytes. The bytes of a storage unit can be read and written in any order. If storage unit contains less than the maximum number of bytes, then the bytes used need not be contiguous. For instance, a storage unit could contain bytes numbered 0 through 100 and 200 through 299, but none of the bytes numbered 101 through 199. The contiguous regions of a storage unit are called **blocks**.

Thus, reading the bytes of a storage unit cannot raise a “end of file” exception since there is no beginning or end to the storage unit. A “non-existent storage” exception would be raised instead. Because the storage unit does not consist of a single contiguous region of storage, it does not have a size either.

In a high performance implementation, the server accesses the disks directly to implement the storage units. For this semester, the storage units are implemented on top of the Windows NT file system. Each storage unit is represented as a pair of standard Windows NT files. One file contains the data of storage unit, with the gaps between the blocks removed. This file is called the **data segment** of the storage unit. The second file contains the information required to reconstruct the gaps and sparsity of the storage unit as well as any locking information. We call the second file the **index segment** of the storage unit.

The **index segment** consists of a sequence of 64 byte records that describe each primitive block. The format of the records in the index segment is shown in Table 1.

The **start** field of the record is the storage unit address of the first byte in this block. The number of bytes in this block is given by the second field. The last two fields are used for locking. We use the simplest possible locking model here. A user with a certain set of credentials can lock a block of storage for exclusive access. The locker field gives the credentials of the locker. The **Lock Date** Field indicates when the lock was estab-
lished. The format of the Lock Date field is the number of milliseconds since the Midnight GMT, January 1, 1970.

If the client wants to lock a range of bytes that is a subset of a block, then the system divides the block into two blocks. One of these blocks is then locked. When the sub-block is unlocked, the server should combine the two sub-blocks to produce a larger block. These operations do not require any operations on the data portion of the file.

<table>
<thead>
<tr>
<th>Field</th>
<th>Size</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Storage ID Start</td>
<td>8 bytes</td>
<td>StorageID address for first byte contained in this block.</td>
</tr>
<tr>
<td>Physical Start</td>
<td>4 bytes</td>
<td>Offset in this file for the first byte in this block.</td>
</tr>
<tr>
<td>Length</td>
<td>4 bytes</td>
<td>Size of block.</td>
</tr>
<tr>
<td>Locker</td>
<td>8 bytes</td>
<td>Credentials of locker</td>
</tr>
<tr>
<td>Lock date</td>
<td>8 bytes</td>
<td>Time lock was established</td>
</tr>
</tbody>
</table>

Table 1. Index File Record Format

For simplicity in this implementation, the server will maintain all of the index segment information in memory and will write out a new index segment file when changes are made.

One additional file is included in the directory used by the storage server. This file maintains any additional information required by the Storage server that is not specific to a Storage Unit. This file is named “StorageServer.def”. At the moment this file is 16 bytes long consists of two 64-bit integers. The first integer is the smallest unused data StorageID and the second is the largest unused directory StorageID.

4.1 Java Implementation of Server

The first implementation of the server was in Java and is documented here. Because we could share a fair amount of the network code between the server and the client implementations, it made sense to implement both the client and the server in Java initially. In later implementations, the server could be implemented in C/C++ for improved performance.

The structure of the Java implementation is shown in above figure. The class SimpleStorageServer implements the semantics of the shared storage system using the WindowsNT file system as described above. This class implements the StorageServer
interface, which is essentially a procedural version of the WireAPI. On the client side, the `UDPStorageServer` class also implements the `StorageServer` interface. However, the methods of `UDPStorageServer` merely translate their arguments into UDP packets that follow the WireAPI protocol. On the server side, the `ServerUDP` class is a wrapper that converts the incoming UDP packets into the `StorageServer` interface. Together, the `UDPStorageServer` and `ServerUDP` classes implement a remote procedure call mechanism for the `StorageServer` interface.

Detailed documentation of the classes and methods used to implement the Shared Storage Server are given in JavaDoc format on the web page. Here we give an overview of the architecture of the system.

The `SimpleStorageServer` class is the heart of the server implementation. Storage units on disk are represented by instances of the classes `DataStore`, for data storage units, and `DirectoryStore`, for directory storage units. Both of these classes are subclasses of the `StorageUnit` class.

Recall that the current implementation of the shared storage server uses WindowsNT files to represent data storage units. One file, the “data segment”, contains the data of the storage unit, while the “index segment” describes how the mapping between the contents of the data segment and storage unit address space. The instance of `DataStore`, which are used to represent a data storage unit, contains all of the information in the index segment, but none of the information in the data segment. The index segment is represented as a list of `BlockRecord`’s, one for each block in the data segment. `BlockRecords` do not contain the data in each block, just where to find the data in the “data segment” of the storage unit. Each time the index segment needs to be modified, a new index segment is generated from the `BlockRecords` contained in the `DataStore`.

Directory storage units are represented as instances of the class `DirectoryStore`. `DirectoryStore’s` maintain the association list between `StorageID’s` and Unicode names in memory, and write this information to the directory storage unit’s index file when necessary.

The WireAPI defines a bit pattern for the different messages that can be exchanged between the client and server processes of the Shared Storage Server. These bit patterns can be transmitted using any network transport protocol, UDP, TCP, or SNA, for instance. To insulate the storage server code from the details of these different protocols, the client and server interact through instances of the `Message` class. Each instance corresponds to a particular packet as described in Section 3. The implementations of the transport protocols are responsible for translating these `Message` objects into patterns for their transport protocol. A subclass of `Message` is defined for each type of packet given in Section 3. This hierarchy of classes is described in Section 5.3.

Basic classes are provided to represent `StorageID’s`, credentials, the exceptions generated by the shared storage server and a few other types of objects. These are described in the JavaDoc documentation and in Section 5.

### 4.2 C Implementation of Server

To be provided in a future implementation.
5. Common Java Classes

The first class that is needed is an object that represents a storage identifier. A class is used for this purpose instead of a long to make the code clearer, easier to debug, and most importantly to prepare for the day when StorageID’s are more complex than 64 bit integers.

```java
public class StorageID {
    long storageID;
    public StorageID(long id);
    public boolean equals(Object ob);
    public StorageID next();
}
```

Credentials are also long integers that are encapsulated in a class. When credentials are actually used, they will probably need to larger structures.¹

```java
public class Credentials{
    // Internal state
    public Credentials(long data);
    public String toString();
    public long toLong();
}
```

5.1 StorageException

All exceptions generated by the storage server, both on the client and server side, are instances of the class StorageException. This class also contains, as static members all of the error codes that are generated by the storage server.

```java
public class StorageException extends Exception {
    private int errCode;
    static public final int IOError = -1;
    static public final int BadWireAPI = -2;
    static public final int CreateDataFail = -3;
    static public final int CreateDirFail = -4;
    static public final int DeleteFail = -5;
    static public final int ReadDataFail = -6;
    static public final int BadBlockAlloc = -7;
    static public final int UnitMismatch = -8;
    static public final int NotStorageID = -9;
    static public final int IndexFNF = -10;
    static public final int DeleteNEexistSU = -11;
    static public final int Unimplemented = -255;

    public StorageException(int i);
    public int getErrorCode();
    public String getMessage();
}
```

The getMessage method will return the a human readable string describing the error condition that has arisen.

¹ The long slot that is used by Credentials should be replaced by a byte array. This would allow more flexibility in the specification of the Credentials fields.


5.2 **StorageServer Interface**

The *StorageServer* interface is a procedural version of the WireAPI that should be obeyed by all implementations of the Shared Storage Server.

```java
package il.ac.idc.storage;

public interface StorageServer {
    /** Create a data storage unit on this Storage Server that can contain at least size number of bytes of data */
    public StorageID createUnit(long size)
        throws StorageException;

    /** Create a directory storage unit on this Storage Server that can contain at least size number of bytes of data. */
    public StorageID createDirectory(long size)
        throws StorageException;

    /** Delete a storage unit. Returns the acknowledgement code */
    public int deleteUnit(StorageID id);

    /** Reads data contained in the indicated storage unit and returns it as an array of bytes. The number of bytes read may be less than that requested. If an error occurs, an exception is thrown. */
    public byte[] readData(StorageID unitID, long start, long len)
        throws StorageException;

    /** Writes the indicated data into the storage unit at the indicated point. The first len bytes in data are written. The value returned is the return value of the acknowledgement. */
    public int writeData(StorageID unitID, long start, long len, byte[] data)
        throws StorageException;

    /** Read an attribute of the storage unit */
    public byte[] readAttribute(StorageID unitID, int Attribute)
        throws StorageException;

    /** Writes an attribute of the storage unit. The return value is the return value of the acknowledgement. */
    public int writeAttribute(StorageID unitID, int Attribute, byte[] data)
        throws StorageException;

    /** directory should refer to a directory storage unit. The directory is searched for the name unitName and, if found, the corresponding StorageID is returned. */
    public StorageID lookup(StorageID directory, String unitName)
        throws StorageException;

    /** Binds the storage unit to a string name in the directory storage unit. */
    public void bind(StorageID directory, String unitName, StorageID unit)
        throws StorageException;

    /** Unbinds a string name in the directory storage unit. */
    public void unbind(StorageID directory, String unitName)
        throws StorageException;

    /** Locks the region of the storage unit for exclusive use by the current
```
client. The acknowledgement value is returned. */
public int lock(StorageID unit, long start, long length);

/**  Unlocks the region of the storage unit for exclusive use by the
  current client. The acknowledgement value is returned. */
public int unlock(StorageID unit, long start, long length);
}

5.3 Storage Server Message Objects

The Java implementation of the shared storage server encapsulates the packets that
are sent between clients and servers in objects that are subtypes of Message. Lower lev-
els of the storage system are responsible for translating these Message’s into the bit pat-
terns required for UDP, TCP, or other transmission protocols.

6. The Client Side

The client side exists at three different levels. The Kernel level provides the low-level
interface to the network and the Storage System protocol defined in Section 3. The Lib-
rary level provides semantically richer interfaces to the storage system. Examples of
these interfaces are a easy to use, Unix-style file system interface and transaction sys-
tem that could be used to build databases and used in the ATM project. The Application
level is used for Web based applications that use the storage system, either directly or
through one of the library interfaces.

6.1 Kernel Level

The kernel of the client side provides a direct implementation of the user side of the
protocol given Section 3. Below we provide the interfaces for the different aspects of the
protocol. Providing the Java implementations of these interfaces should be a good exe-
cise for the students.

From the client’s perspective, an object (in the client’s address space) that fronts from
the real storage server process represents a storage server. The class of all such objects
implements the il.ac.idc.storage.client.StorageServer interface, which is given below.

We provide one implementation of the client’s StorageServer interface,
UDPStorageServer. Instances of this class can be created by invoking the classes creator
method:

public UDPStorageServer(InetAddress address, Credentials cred)
throws StorageException;

Once an instance of UDPStorageServer has been created, operations on the storage
server can be effected by calling the methods of the StorageServer interface documented
in Section 5.2.

6.2 Library Level

The interface provided by the abstract class in Section 6.1 is very flexible and can be
quite efficient, but it is difficult to use and does not match the higher level semantic
models with which programmers are usually familiar. The library level provides inter-
faces that are easier to use and are a closer semantic match to applications. These inter-
faces...
faces are not currently provided in the shared storage server, but they are documented here for future implementations.

In Section 6.2.1, we discuss a file-oriented interface along the lines of the Unix file system. Section 6.2.2 describes a simple database built using the shared storage server.

6.2.1 Unix Style File System Interface

The following file system interface is basically that of Unix. A file is a contiguous sequence of bytes. The File interface captures the concept of an open file.

```java
Package il.ac.idc.storage.fs;
public interface File {
    /** Write the given byte array to this file. Successive writes will cause the
     * concatenation of the contents. If the file is not open, or the write
     * fails for some other reason an exception will be thrown.
     */
    public void write(byte[] a) throws FileException;

    /** Read from the file into the given byte array. The array will be filled to
     * its length, unless an EOF was encountered. The number of bytes
     * read is returned.
     */
    public int read(byte[] a);

    /** Move the current position in the file offset bytes forward. Offset can be
     * positive or negative. Returns the resulting position in the file.
     */
    public long seek(long offset);

    /** Release the file. */
    public void close();
}
```

A file system is a collection of files, which have names that are strings. A file system is represented by classes that implement the FileSystem interface given below. To represent file systems with directories, the class should also implement the Directory interface, which is given later.

```java
public interface FileSystem {
    /** Open a file for reading */
    public File openRead(String name);

    /** Open a file for writing. If it exists then it gets re-written. Otherwise,
     * the file is created. */
    public File openWrite(String name);

    /** Delete a file */
    public void delete(String name);

    /** List the existing files */
    public String[] list();
}
```

File systems that have directories can implement hierarchical structure. The following interface describes the functionality of a directory object.

```java
public interface Directory {
    /** Create a storage unit that represents a file and associates it with
     */
    public void create(String name);
}
```
given filename. The StorageID of the storage unit created is returned. A StorageException will be thrown if there it is not possible to create the requested storage unit for some reason. */
StorageID createFile(String fileName)
    throws StorageException;

/** Create a storage unit that represents a file and associates it with
given filename. The StorageID of the storage unit created is returned. A StorageException will be thrown if there it is not possible to create the requested storage unit for some reason. */
StorageID createDirectory(String directoryName) throws StorageException;

/** If there exists a storage unit with this name in the directory, then
delete it. If we do not permit multiple references to the same storage
unit then delete the storage unit as well. If multiple references to the
storage unit are permissible, then only delete the underlying storage
when the last reference to a storage unit is deleted. */
void delete(String name)
    throws StorageException;

/** Bind a storage unit to a name in this directory. Multiple references to
the same storage unit, and circularities can easily be created with this
method. */
void bind(StorageID unit, String name)
    throws StorageException;
}

6.2.2 Simple Database

7. Java Implementation Notes

This section provides additional documentation on the implementation provided on the course web site, the test routines, etc.

On the handout page of the web site you will find a zip file that contains a single directory containing the sources for the shared storage server, its documentation, and the Visual Studio Workspace for the project. To use these

7.1 Development Environment

This document only discusses the the Microsoft Visual Studio development environment. You should be able to use Sun’s Java Workshop in much the same manner as VisualStudio, but I haven’t been able to set up the project files properly yet. Project files for Java Workshop are present in the hierarchy if you would like to start from there.

7.1.1 Microsoft Visual Studio

You may be using one of three different versions of Microsoft’s J++ environment. If you installed VJ++ 6.0 from the course web site, then the compiler and environment you have should be plenty to support your work in this course. However, all of the examples here are based on VJ++ 1.1, so you are on your own to a certain extent.

If you installed VJ++ 1.0, perhaps the student edition that comes with Deitel and Deitel, then the following instructions should work, but I am not checked it to make sure.
If you are using Visual J++ 1.1, you should apply the Java SDK (provided on the course web site, or at Microsoft) to bring it up to JDK 1.1 compliance. These needs to be done in several steps.

First download from the course Web site, the files SDK-Java.zip, MSJavx86.zip, javadocs.zip and sdkjdoc.zip. If you want to minimize the space used on your hard disk, you only need to deal with the first two files. The last two zip files contain documentation. Next, unzip the files that you have downloaded to produce .exe files. Run SDK-Java.exe to install the SDK, and then MSJavx86.exe to install the latest version of the Microsoft Java virtual machine. At this point you will need to reboot your computer to finish the installation.

At this point you still need to install the JDK 1.1 compiler as well. To do this, first locate the current version of your Java compiler. Most likely it will be “C:\ Program Files\DevStudio\SharedIDE\bin\jvc.exe”. Rename this file to something else (e.g., jvc-old.exe). Next copy the three files jvc.exe, msjvc.dll, and jps.dll from C:\SDK-Java.30P1\bin to the directory with your old version of jvc, typically “C:\Program Files\DevStudio\SharedIDE\bin” directory.

Once your development environment is setup you can try loading the Shared Storage Server workspace into J++. Either double click on SYP/syp.dsw, or load that workspace into a running version of the Developer Studio. When you do, you should get a screen that looks like that shown in Figure 2.

Notice that the workspace contains nine separate projects. The projects Client and Server contain the code for the client library and the server itself. In addition the Common project contains many Java class that are common to both the client and server. The
Database and FileSystem projects will be used in the future. The TestClient and TestServer projects contain test programs that were used when developing the Shared Storage Server.

The project that is open, VirtualArray, can be used for your work in problem set 2. The VirtualArray project depends upon two other projects (Common and Client), and contains three files. In addition the CLASSPATH should be correctly set up that it contains two subprojects and three files. Although it should be well set up we will walk through the steps of setting up the project to help you when doing your own development.

First, we want to make sure that VirtualStorage depends upon the Client and Common projects. When a project appears as a subproject of another in the File view, it means that the subprojects will be kept up to date when the project itself is built. To ensure that VirtualArray “depends on” Client and Common, you should select the Dependencies item of the Project menu. This should give you a menu that looks like the following. Notice that if other projects are clicked they will be added as subprojects of VirtualArray.

![Project Dependencies](image)
Next we need to make sure that the compiled class files go to the proper place. This can be achieved by selecting the *Project Settings* item from the *Project* menu. This will give you a menu that looks like the following: The Output Directory field needs to be ..\lib so that the class files will be placed there.

Finally, you must select the Debug tab, and make sure that the correct version of jview is used to run the stand alone application. You will get a window that looks like the following. The version of this project that is provided on the web server looks in C:\winnt\ for jview. If you are not running on an WindowsNT machine, you probably need to change winnt to windows, win95 or something similar.
7.2 **Running the Shared Storage Server**

To start the shared storage server, you first need to allocate a directory for the Storage Units. I often use `C:\temp\test` for this purpose, although any directory will do well. This directory must be created, although it can be empty before you proceed to the next step. Open an MSDOS window and change to the `SYP/lib` directory. Then start the Shared Storage Server by typing `jview ServerStart C:\temp\test`. This should give you a result like the following:

![Command Prompt - jview ServerStart C:\temp\test](image)