University of Tennessee, Knoxville

TRACE: Tennessee Research and Creative Exchange

Chancellor's Honors Program Projects

Supervised Undergraduate Student Research and Creative Work

Spring 5-2002

Building a Content-Addressable IBP Server

Rebecca Lynn Collins

University of Tennessee - Knoxville

Follow this and additional works at: https://trace.tennessee.edu/utk_chanhonoproj

Recommended Citation

Collins, Rebecca Lynn, "Building a Content-Addressable IBP Server" (2002). *Chancellor's Honors Program Projects.*
https://trace.tennessee.edu/utk_chanhonoproj/526

This is brought to you for free and open access by the Supervised Undergraduate Student Research and Creative Work at TRACE: Tennessee Research and Creative Exchange. It has been accepted for inclusion in Chancellor's Honors Program Projects by an authorized administrator of TRACE: Tennessee Research and Creative Exchange. For more information, please contact trace@utk.edu.

UNIVERSITY HONORS PROGRAM

SENIOR PROJECT - APPROVAL

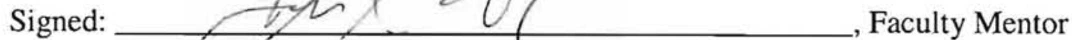
Name: Rebecca Collins

College: Arts and Sciences Department: Computer Science

Faculty Mentor: Dr. James Plank

PROJECT TITLE: Building a Content-Addressable IBP Server

I have reviewed this completed senior honors thesis with this student and certify that it is a project commensurate with honors level undergraduate research in this field.

Signed:  , Faculty Mentor

Date: 5/2/03

Comments (Optional):

The following documents are included in this packet:
As of May 1, 2003, the two papers are available at http://www.cs.utk.edu/~rcollins

Architecture of the Content-Addressable IBP (IBPCA)

- describes the architecture of IBPCA
- includes results of performance testing of IBPCA compared to IBP

IBPCA: IBP with MD5

- describes the client API for IBPCA

Code for the IBPCA

- ibpca_server.c
- ibpca_server.h
- ibp_ca.c
- ibp_ca.h
- ibpca_io.c
- ibpca_io.h

Architecture of the Content-Addressable IBP (IBPCA)

Rebecca Collins

Abstract

This document describes an extension to the Internet Backplane Protocol (IBP), called IBPCA. While IBP provides the ability for clients to manage and employ remote storage resources, IBPCA adds the ability for clients to address storage by hashes of the storage's contents. Thus, it adds a content-addressable layer on top of IBP. This document describes the functionality and architecture of IBPCA, and addresses its performance implications as compared to unenhanced IBP.

1 Introduction

IBP is a system for managing remote storage over a network. IBP servers perform this function by allowing clients to allocate space on a depot, and move data to and from a depot as well as between depots. The storage management that IBP provides can be thought as buffer management, where files sent over the Internet are buffered in a temporary byte array, or as a network resource for more permanent files for users to share like they share bandwidth [2].

IBP operates as server daemon software and a number of procedure calls for the client API. The IBP server makes local storage available for remote management. To highlight the logistical nature of servers, we call them depots, drawing an analogy to material transportation systems. IBP is designed with optional restrictions on resource usage; for example, the server may be restricted to using idle memory only, or enforcing a time-limit on all storage allocations [2].

Clients are able to access byte arrays in IBP depots by means of capabilities. There are three types of IBP capabilities: read, write, and manage. Read capabilities give the client access to load the contents of the byte array, write capabilities give the client access to append to the contents of the byte array, and manage capabilities give the client access to probe and change aspects of the byte array, like duration and size. Capabilities in IBP come in the form of ASCII text strings that may be passed around the network easily:

```
ibp://hostname:port/key/WRMKey/WRM
```

MD5 is a message digest algorithm used to generate a 128-bit hash from a data file. Files that differ by one or more bits typically have different hashes, but with a very small probability, the hashes can be the same. This is called a collision. The contents of a file cannot be determined from its MD5 hash [1]. In this document, MD5 hashes are also referred to as checksums.

IBPCA adds content addressibility to IBP by incorporating MD5 hashes of the stored data into the capabilities. The hashes can be used to index and verify stored data.

The benefits of having the MD5 hash of stored data in the read capabilities are that the client can verify that data is transferred intact over the network, and the server can know if it is receiving requests to store the same data twice. For the server, this is an advantage because redundant storage and network transfers may be avoided.

2 IBPCA: IBP with MD5

2.1 IBPCA Architecture

As a prototype implementation of content-addressibility, we chose to implement the IBPCA functionality in a separate server, co-located with an unaltered IBP server. The IBP server manages the IBP depots exclusively, while the IBPCA server handles the bookkeeping details of conversions between IBP and IBPCA capabilities and acts as an intermediary for most communications between the client and the IBP server. The reason for the existence of IBPCA is to make the data stored in IBP depots content addressable. The IBPCA client library usually connects to the IBP server through the IBPCA server but in some API calls it communicates directly with the IBP server.

2.2 Client API

There are five operations currently implemented within IBPCA. **Figure 1** provides a definition of the five procedure calls along with a brief description of what each call does.

2.3 Capabilities

In IBP, clients use read, write and management capabilities to manipulate byte arrays. When a client allocates space, all three capabilities are returned to the client. In IBPCA, clients use only read and write capabilities. The capabilities in IBPCA have the following format:

```
Read capabilities: ibpca://hostname:port/R/MD5-checksum
Write capabities: ibpca://hostname:port/W/random_string
```

where:

- `hostname` and `port` are the hostname and port of the IBPCA server
- `MD5-checksum` is the MD5 hash of the data in the storage depot that the read capability references
- `random_string` is a randomly generated 32 character string

The write capability is returned after allocation, but since the `MD5-checksum` part of the read capability is computed from the contents of the data being stored, the read capability is not returned until after data has been stored. In addition, the contents of a byte array change after each append, so a new read capability must be generated after each append. All read capabilities that correspond to incremental stores in a given byte array continue to be valid for the duration of the byte array.

Saving old read capabilities works because byte arrays stored through IBPCA are append-only. Thus, if n bytes are appended to an empty byte array, then the read capability will be able to load the contents of the first n bytes of the byte array. If any new data is appended to the byte array, then a new read capability with the new checksum will be required for accessing the entire byte array, but the original read capability will still be sufficient for accessing the first n bytes of the byte array.

Instead of using management capabilities, IBPCA clients use a read capability or a write capability to manage a byte array. Providing a single manage capability is insufficient because the byte array can have several different read capabilities, all with different sizes, and the size of a given read capability cannot be increased because the read capability would then require an updated MD5 checksum. For this reason, there are no manage capabilities in IBPCA.

```
IBP_CA_allocate(      IBP_depot depot, int maxsize,
                      IBP_cap writecap);
```
- allocates a byte array in an IBP depot;

```
IBP_CA_store(         IBP_cap ca_writecap, char *data,
                      ulong_t size, IBP_cap readcap);
```
- appends data to an allocated byte array;

```
IBP_CA_load(          IBP_cap readcap, char *buf, int
                      size, int offset);
```
- retrieves data from a depot;

```
IBP_CA_manage(        IBP_cap man_cap, int cmd,
                      ibp_probe_info *status);
```
- manages a byte array (e.g. extends duration, increases size); Note: `man_cap` is either a read capability or a write capability

```
IBP_CA_store_block( IBP_depot depot, int size, char *data,
                      IBP_cap ca_readcap);
```
- stores a block of data that will have read-only access

Figure 1: IBPCA Client API (Note: Error and bookkeeping details have been omitted for clarity; the full API is available in the Appendix.)

3 Organization

3.1 Bookkeeping

When space is allocated on an IBP depot, IBP returns read, write, and management capabilities for that storage space. The IBPCA server returns a write capability after an `IBP_CA_allocate` call and a read capability after every `IBP_CA_store` call. IBPCA's main purpose is keeping track of the IBP capabilities and IBPCA capabilities that correspond to each other. To do this, the IBPCA server maintains two directories—one for read capabilities, and one for write capabilities. The directories are red-black trees that are described in Table 1.

Directory	IBPCA Readcap Directory	IBPCA Writecap Directory
Key	IBPCA read capability	IBPCA write capability
Value	IBP read capability, IBP manage capability, size, duration	Set of IBP capabilities: read, write and manage, size of byte array, current size that is utilized, duration, MD5 context (needed to update MD5 checksum when new data is appended to a byte array)
Size	2437 bytes	3549 bytes

Table 1: Readcap and Writecap Directories (some of the more trivial details are omitted)

A list of durations of the `ca_read` and `ca_write` capabilities is also maintained so that capabilities may be deleted when they have expired. The memory taken up by the read and write capabilities in the duration list is included in the `size` element of **Table 1**.

3.2 Communication

The three communicating parties are the IBPCA server, the IBPCA client, and the IBP server. The IBPCA server operates on a level above the IBP server using the IBP API (in effect it is a client to the IBP server). Since the servers operate separately, the IBPCA server has no access to the internal IBP structures. For this reason some information like size and duration of a byte array must be recorded twice.

IBPCA provides the client API defined in **Figure 1**. The IBPCA API substitutes the IBP API for the client. Internally, IBPCA uses the IBP API. In cases where data is not being transferred, all IBP API calls come from the IBPCA server. In cases where data is being transferred, the IBP API calls come from the IBPCA client, bypassing the IBPCA server so that the data is not sent over the network twice.

Compared to IBP, IBPCA involves an increased number of communications between the client and server. This increases the probability of failure in the applications due to network or other complications. This aspect of IBPCA was not considered in the design, and has not been tested.

The rest of this section provides a summary of communication paths of each API call.

IBP_CA_allocate

The `IBP_CA_allocate` call first connects to the IBPCA server. Then the IBPCA server connects to the IBP server with an `IBP_allocate` call. The IBP server returns capabilities to the IBPCA server and the IBPCA server returns a content addressable write capability to the client through `IBP_CA_allocate`.

IBP_CA_store_block

The `IBP_CA_store_block` call first connects to the IBPCA server. Then the IBPCA server connects to the IBP server with an `IBP_allocate` call. The IBP server returns capabilities to the IBPCA server and the IBPCA server sends the IBP write capability to the client. The client connects to the IBP server with an `IBP_store` call. The client sends a message to the IBPCA server describing the success of the call, and the server returns a content addressable read capability to the client.

IBP_CA_store

The `IBP_CA_store` call first connects to the IBPCA server and sends over the content addressable write capability. The IBPCA server then sends the IBP write capability corresponding to the client's content addressable capability and the client connects to the IBP server with an `IBP_store` call. The client lets the IBPCA server know if the call was successful, and if so, the IBPCA server returns a content addressable read capability to the client.

IBP_CA_load

The `IBP_CA_load` call first connects to the IBPCA server and sends over the content addressable read capability. The IBPCA server then sends the IBP read capability corresponding to the client's content addressable capability and the client connects to the IBP server with an `IBP_load` call.

IBP_CA_manage

The `IBP_CA_manage` call first connects to the IBPCA server and sends over a content addressable capability. The communication path then varies according to the manage command specified by the client in the call. If the manage parameter is IBP_CA_MANAGE_TIME or IBP_CA_MANAGE_SIZE, the IBPCA server connects to the IBP server with an `IBP_manage` call and the IBP_CHNG parameter. If the manage parameter is IBP_CA_MANAGE_PROBE, the IBPCA server returns the size and duration of the content addressable capability. If the capability is a write capability, the maximum size of the byte array is returned, while if the capability is a read capability, the size of the data accessible with the read capability is returned. If the manage parameter is IBP_CA_MANAGE_DEL, the IBPCA server removes the write capability from its directories.

4 Advantages of IBPCA

The advantages of having a content addressable IBP are that clients have the ability to check the integrity of stored data, and that the server can save network bandwidth and disk space on the depot through detection of duplicate files.

4.1 IBPCA Verifies that Stored Data is Unchanged

The client can check the integrity of stored data since the MD5 checksum of a file should be the same before and after the file is stored in the IBP depots. The checksum of the downloaded data can be compared to the original checksum from the read capability.

4.2 IBPCA Avoids Unnecessary Data Transfer and Storage

When a client wants to store a file, the checksum of the file is compared to the checksums of files that are already stored. If the file with that checksum already exists in the depot, then the client just gets access to the existing storage and the data is not retransferred. This saves both network bandwidth and disk space on the depot.

4.3 The Contents of Depots are Visible through the Capabilities

Since the only information in an IBPCA read capability is the host name, port, and MD5 checksum of the stored data, a client is capable of constructing a valid read capability and then querying the depot to see if data is there.

5 Disadvantages of IBPCA

The main disadvantage of the IBPCA is that it requires extra storage and computational overhead in addition to the overhead incurred by the IBP. Second, using MD5 hashes to reference stored data will fail when two different files have the same hash.

5.1 Overhead

The storage and computational overhead of storing data on the IBP depots will increase with the use of the IBPCA. Using IBPCA on top of IBP means that basic information like duration and size of the storage will be stored twice. In addition, the IBPCA requires an extra 2.5 KB of overhead storage every time an append is made. The extra time IBPCA functions will take in comparison to IBP functions is discussed in section 6.

5.2 Checksum Collision

Although MD5 is designed so that it will generate different hashes for different files, there can be at most 2^{128} hashes. Since there can be infinitely many different files, at least one hash is shared by multiple files. If there is ever a collision in the MD5 hashes of two different files that are sent to the IBPCA through `IBP_CA_store_block`, then the second one will not be stored, and the client storing the second file will receive read capabilities for the first file. However, while a collision is possible, it is highly unlikely to happen by chance, and it is currently computationally infeasible to deliberately create a file with a given hash [1].

6 Performance Case Studies

To test the time penalty that IBPCA introduces when it is used instead of IBP, two sets of performance tests were conducted. The first set tested IBP and IBPCA over a local network where both servers ran on a dual-processor 450 MHz UltraSPARC-II machine with 2 Gbytes RAM and the client ran on a Sun Blade 100 workstation with one 500-MHz UltraSPARC-IIe processor, 512-Mbytes RAM and a 100 Mbps switched Ethernet connection. Both machines were part of the Computer Science's network at the University of Tennessee in Knoxville, Tennessee. The second set tested IBP and IBPCA over a wide network where the client ran on the same machine in Knoxville, and the servers ran in San Diego on a machine with an AMD Athlon XP 2100+ 1733 MHz processor and 512-Mbytes RAM.

For load, store, and store_block, the server performance was tested for five different data sizes: 1 byte, 1 KB, 1 MB, 3 MB, and 20 MB. The 1 KB, 1 MB and 20 MB data consisted of compressed data files, and the 3 MB file was an MP3 file. The data sizes stated above are approximate; the exact sizes are listed in **Table 2**. In many cases, the individual call durations were too short to be recorded, so durations were taken as averages of multiple function calls. Each data point represented in the following charts is the average of at least 10 such values.

Approximate Size	Actual Size
1 byte	1
1 KB	1001
1 MB	1066377
3 MB	3026156
20 MB	21230657

Table 2: Exact Data Sizes

6.1 Allocate

The results of the tests for `IBP_allocate` and `IBP_CA_allocate` are shown in **Figure 1**. In the tests over a local network, IBP performed about 5 times better than IBPCA, but in the tests over a remote network, the IBP performed the same as IBPCA. The actual data shows that IBPCA performed slightly better in the tests over a wide area network, but since `IBP_CA_allocate` actually calls `IBP_allocate`, more extensive testing would probably show that `IBP_allocate` is slightly faster than `IBP_CA_allocate`.

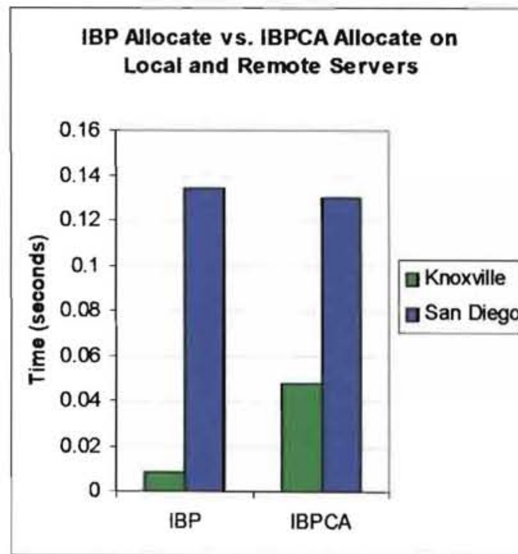


Figure 1: IBP_allocate vs. IBP_CA_allocate

6.2 Load

Figures 2 shows the performance of `IBP_load` and `IBP_CA_load`. **Figure 3** shows the percent increase in IBPCA's performance compared to IBP's performance. For data less than 1 KB, `IBPCA_CA_load`'s duration was 100% longer than `IBP_load`'s duration, but for data greater than 1 MB, the performance of the two functions was almost the same. Like `IBP_CA_allocate`, `IBP_CA_load` actually calls `IBP_load`, so its performance will always be dependent on the performance of the `IBP_load`.

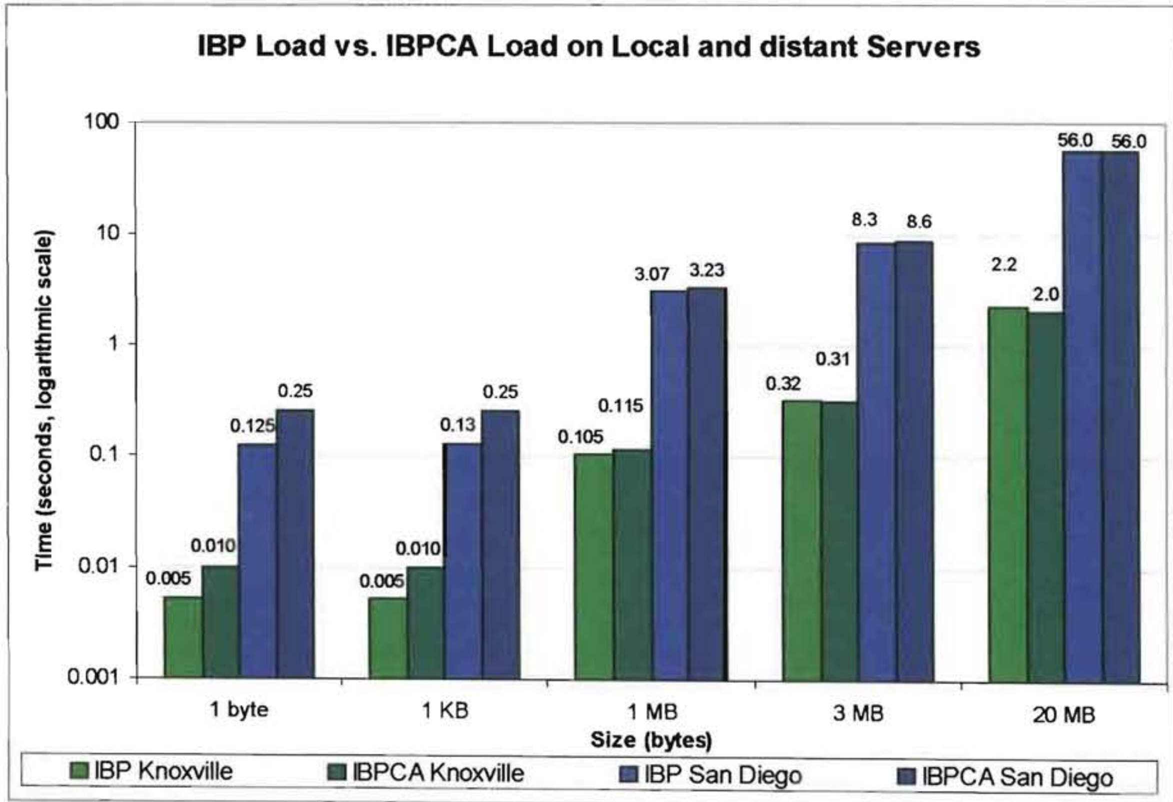


Figure 2: IBP_load vs. IBP_CA_load

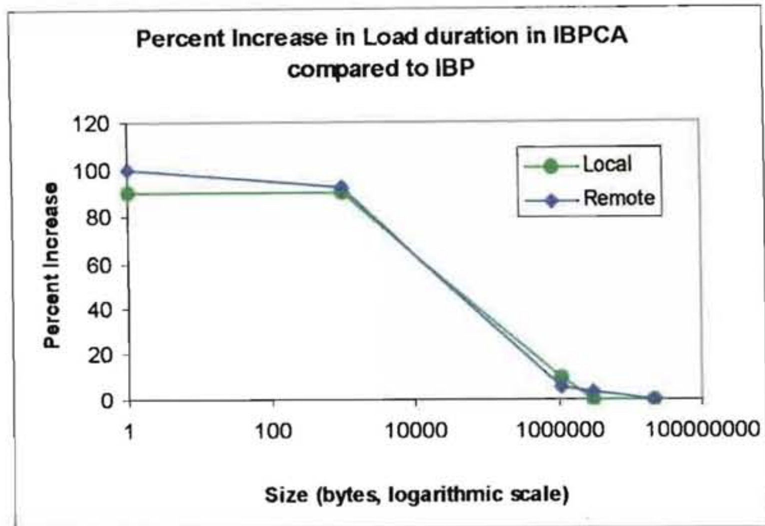


Figure 3: Percent Increase of Load when using IBPCA

6.3 Store

The results of the performance tests of `IBP_store` and `IBP_CA_store` are shown in **Figures 4** and **5**. IBPCA performs significantly slower than IBP over a local network for smaller data blocks (size < 1 MB).

Compared to `IBP_CA_load`, `IBP_CA_store` performed poorly. One reason for this difference in performance is that in `IBP_CA_store`, the client and server have eight total communications while in `IBP_CA_load`, the client and server only have three total communications.

Two of the communications between the client and server in `IBP_CA_store` involve sending an MD5 context back and forth. MD5 contexts hold the MD5 hash information of the data currently stored. When new data is appended to a byte array, the server only needs the context to update the MD5 hash of all of the stored data. First the context of the current stored data must be sent from the server to the client and then the updated context must be sent back to the server. The size of an MD5 context is 88 bytes, so for small data it would be faster to send the new data directly to the IBPCA server and let the IBPCA server update the context itself. Altering the IBPCA server so that `IBP_CA_store` treats small data differently from large data may improve the time performance of `IBP_CA_store`.

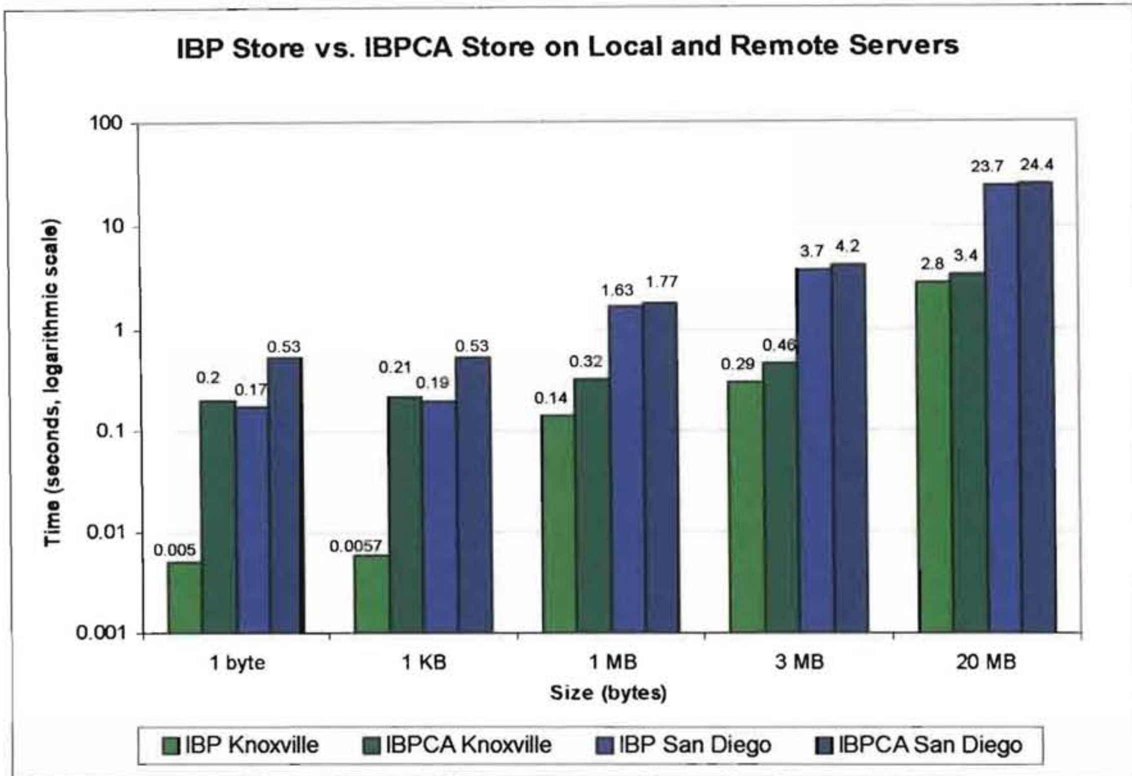


Figure 4: IBP_store vs. IBP_CA_store

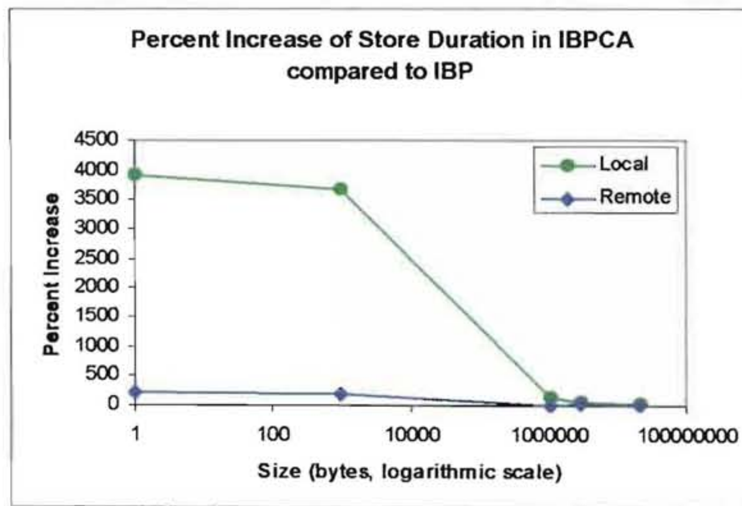


Figure 5: Percent Increase of Store Performance

6.4 Store_Block

The results of the performance tests of IBP_CA_store_block are shown in **Figures 6,7** and **8**. Since there is no equivalent IBP call to IBP_CA_store_block, IBP_CA_store_block was compared to IBP_allocate followed immediately by IBP_store. The special option offered by IBP_CA_store_block is that it can recognize if a block of data has already been stored and avoid sending the data over the network twice. IBP_CA_store_block performance was tested both when data was always being stored for the first time, and when the data was always already stored. Like IBP_CA_store, IBP_CA_store_block had somewhat slow performance for storing small (unique) data blocks over a local network. Using IBP_CA_store_block for data that has already been stored, however, is faster than the IBP alternative. **Figure 8** shows the percent decrease in speed of IBP_CA_store_block when the data being sent to the server is redundant.

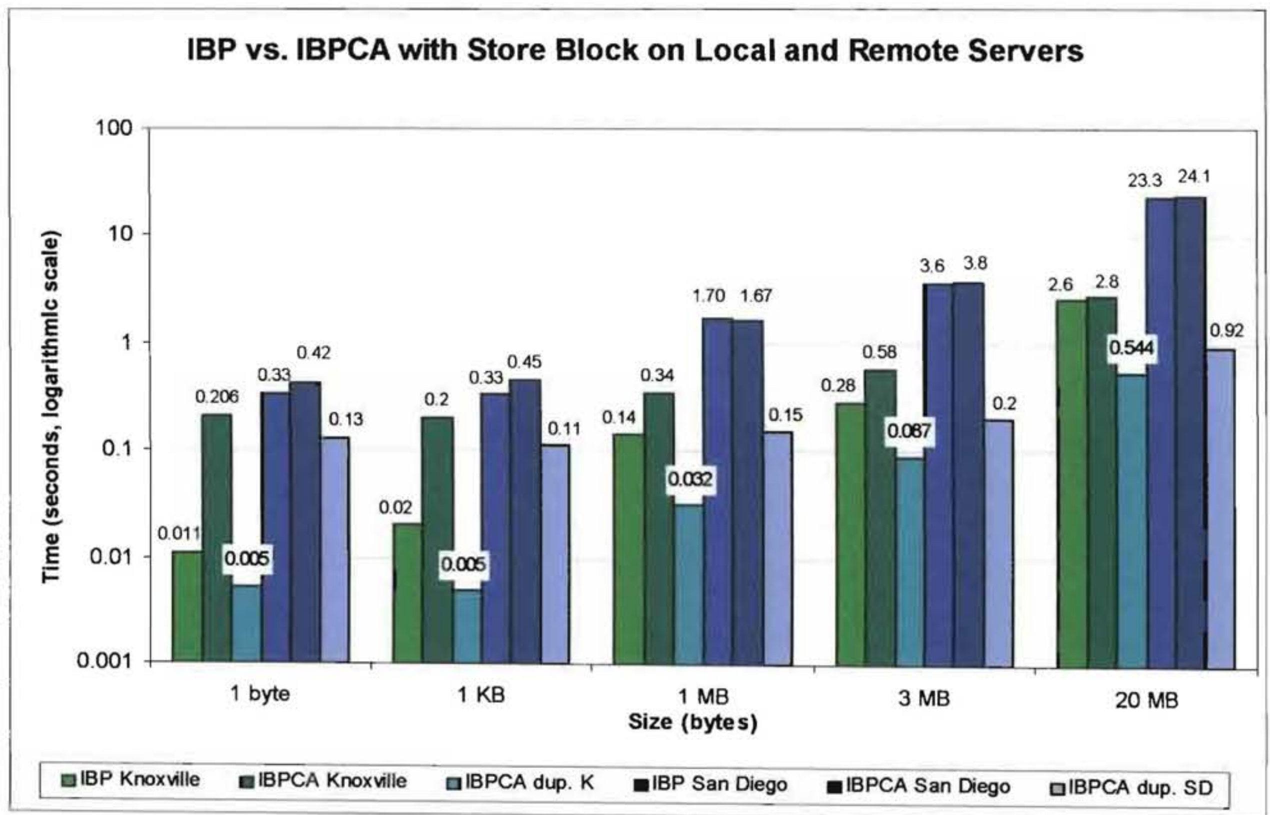


Figure 6:IBP_store and IBP_allocate vs. IBPCA_store_block

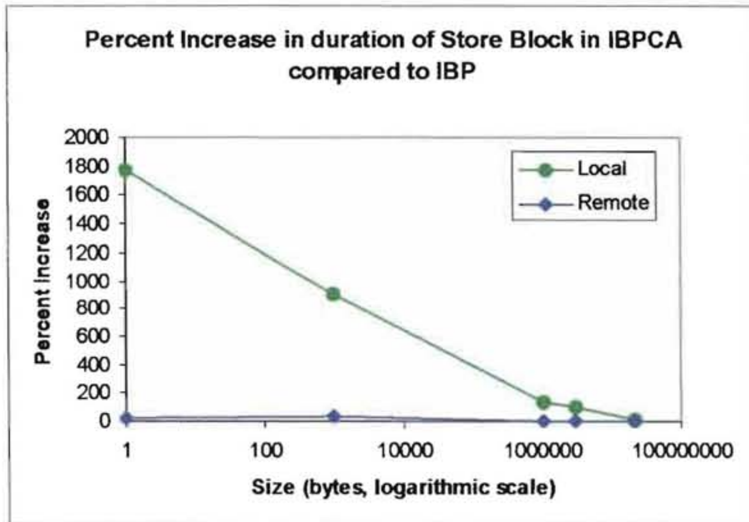


Figure 7: Percent Increase of Store_block

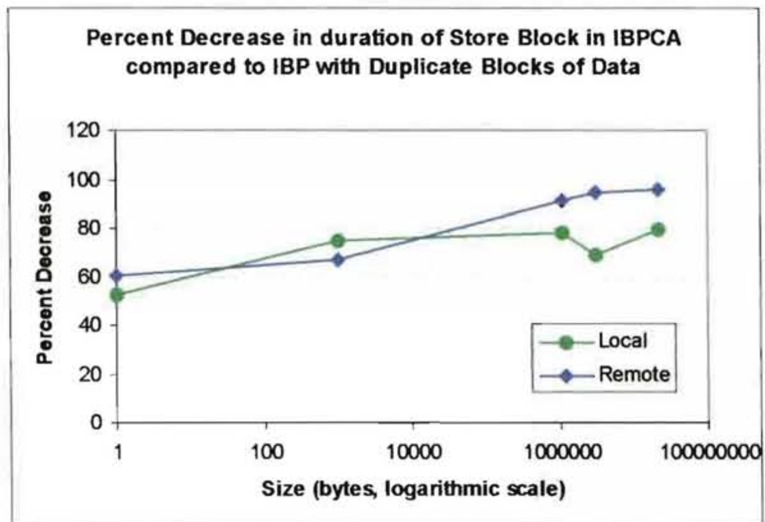


Figure 8: Percent Decrease of IBPCA_store_block's performance compared to IBP when storing redundant data

6.5 Manage

Figure 9 shows the results of the manage performance tests. In these tests, the performance of IBP and IBPCA was fairly similar.

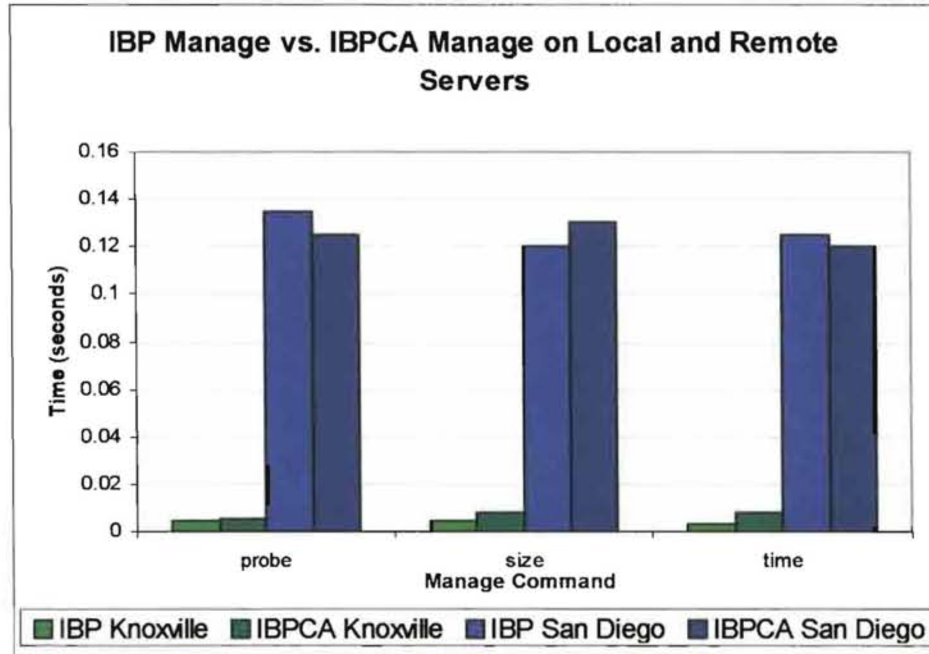


Figure 9: IBP_manage vs. IBP_CA_manage

7 Conclusion

IBPCA introduces a new feature to IBP--the ability to address data by its content. This feature offers users the opportunity to verify the validity of data that has been stored or loaded from the IBP depots and have a moderate improvement in the time that it takes to store duplicate files. The time that it takes to store original blocks of data, however, is significantly increased for small data files. And with its bookkeeping overhead, IBPCA is unviable as a system for saving space on the servers unless it is used with a large number of duplicate files that are greater in size than 2.5 KB.

The performance times of IBPCA were significantly slower on smaller data files than those for IBP especially with the storing functions `IBP_CA_store` and `IBP_CA_store_block`. The increase in execution time of `IBP_CA_load` and `IBP_CA_manage`, on the other hand, was less substantial. Unlike storing and allocating, these functions are more likely to be used multiple times for the same data area, so the overall time penalty introduced by IBPCA is probably not as bad as the `IBP_CA_store` and `IBP_CA_store_block` data implies.

References

[1] Rivest, R. "The MD5 Message-Digest Algorithm", RFC 1321, and RSA Data Security, Inc., April 1992.

[2] J. Plank, M. Beck, W. Elwasif, T. Moore, M. Swany, and R. Wolski, "The Internet Backplane Protocol: Storage in the Network," presented at *NetStore99: The Network Storage Symposium*, Seattle, WA, 1999.

Appendix

IBP_CA_allocate

#include "ibp_ca.h"

int **IBP_CA_allocate**(IBP_depot **depot**, IBP_timer **timeout**, ulong_t **maxsize**,
IBP_attributes **attributes**, IBP_cap **writecap**);

IBP_CA_allocate allocates **maxsize** bytes of storage into the **depot**, with attributes **attributes**. The duration must be finite, and the data type must be byte-array.

Return Values
On successful completion, 0 is returned, otherwise –1 is returned and an error message is sent to stderr. The following conditions will cause IBP_CA_allocate to fail:

- Invalid attributes – For example, infinite duration
- IBP_allocate error – Described in section 3 of IBP v1.1.1 API.

IBP_CA_load

#include "ibp_ca.h"

int **IBP_CA_load**(IBP_cap ca_readcap, IBP_timer timeout, char *buf, ulong_t size,
ulong_t offset);

IBP_CA_load loads **size** bytes, starting at the **offset** position, from the byte-array accessed through **readcap** and stores the bytes into **buf.**

Return Values
On successful completion, the number of bytes read is returned, otherwise –1 is returned and an error message is sent to stderr. The following conditions will cause IBP_CA_load to fail:

- Invalid size/offset
- Readcap has an expired duration
- IBPCA internal error
- IBP_load error – Described in section 4 of IBP v1.1.1 API.

IBP_CA_manage

#include “ibp_ca.h”

int **IBP_CA_manage**(IBP_cap man_cap, IBP_timer timeout, int cmd, ibp_probe_info *status);

IBP_CA_manage lets the user perform the following operations on a byte array (**IBP_MANAGE_SIZE** and **IBP_CA_MANAGE_DEL** require a writecap):

- **IBP_CA_MANAGE_PROBE** updates **status** with information about the byte array: whether it exists (if the capability is valid), its attributes, size, and maximum capacity.
- **IBP_CA_MANAGE_TIME** extends the duration of a byte array unless the new duration is less than the one that already exists.
- **IBP_CA_MANAGE_SIZE** increases the maximum size of a byte array unless the new size is smaller than the one that already exists.
- **IBP_CA_MANAGE_DEL** deletes a write capability

Return Values
On successful completion, 0 is returned, otherwise –1 is returned and an error message is sent to stderr. The following conditions will cause IBP_CA_manage to fail:

- IBPCA internal error
- Writecap/readcap is invalid or expired
- A readcap was sent when a writecap was required
- IBP_manage error – Described in Section 7 of IBP v1.1.1 API.

IBP_CA_store
#include “ibp_ca.h”
int **IBP_CA_store**(IBP_cap ca_writecap, IBP_timer timeout, char *data, ulong_t size, IBP_cap readcap);

IBP_CA_store appends to a **writecap** previously obtained from **IBP_CA_allocate**. The first **size** bytes of **data** are appended to the byte-array referenced by **writecap**. **Readcap** is then constructed from the entire byte-array’s MD5-checksum.

Return values
On successful completion, 0 is returned, otherwise –1 is returned and an error message is sent to stderr. The following conditions will cause IBP_CA_store to fail:

- Expired or Invalid writecap
- IBPCA internal error
- IBP_store error –Described in section 3 of IBP v1.1.1 API.

IBP_CA_store_block

#include "ibp_ca.h"

int **IBP_CA_store_block**(IBP_depot depot, IBP_timer timeout, ulong_t size, char *data, IBP_attributes attributes, IBP_cap ca_readcap);

IBP_CA_store_block allocates and stores **data** into the **depot**. The duration of the storage must be finite, and the data type must be byte-array. The allocated storage space will be read only. If the data to be stored is identical to one already stored, it will not be sent over the network a second time. Instead, the readcap for the original storage will be returned and the duration of the storage will be updated.

Return values
On successful completion, 0 is returned, otherwise –1 is returned and an error message is sent to stderr. The following conditions will cause IBP_CA_store_block to fail:

- Invalid attributes – For example, infinite duration
- IBP_store error – Described in section 3of IBP v1.1.1 API.
- IBP_allocate error – Described in section 2 of IBP v1.1.1 API.

IBPCA: IBP with MD5

Rebecca Collins
rcollins@cs.utk.edu

Abstract

This document presents a description of the five client applications for the Content Addressable IBP (IBPCA). The description includes the data structures and calls used by the C implementation of IBPCA. Knowledge of IBP is assumed.

Introduction

IBPCA uses MD5 hashes to reference data stored in the depots. This is done by incorporating the MD5 hash of stored data in the read capability for a storage area. When new data is appended to a storage area, the updated storage area requires a new read capability. IBPCA operates a level above IBP.

Contents

1. DATA STRUCTURES
1.1 IBP_depot, IBP_timer, and IBP_attributes
1.2 ulong_t
1.3 IBP_cap
1.4 ibp_probe_info
2. CLIENT APPLICATIONS
2.1 IBP_CA_allocate
2.2 IBP_CA_load
2.3 IBP_CA_manage
2.4 IBP_CA_store
2.5 IBP_CA_store_block
3. INPUT/OUTPUT SUMMARY
4. ADVANTAGES OF USING IBPCA
4.1 Checking Integrity of Stored Data
4.2 Preventing Redundant Network Traffic

1. DATA STRUCTURES

1.1 IBP_depot, IBP_timer, and IBP_attributes

IBP_depot, IBP_timer, and IBP_attributes are all defined in IBP_ClientLib.h, a library from the IBP. Descriptions of IBP_depot, IBP_timer, and IBP_attributes can be found in section 1 of IBP v1.1.1 API.

1.2 ulong_t

ulong_t is an unsigned long integer. It is also defined in IBP_ClientLib.h

1.3 IBP_cap

IBP_cap is a char * . It is also defined in IBP_ClientLib.h. IBP_cap's are supposed to have a special format. With IBP, they have the format:

```
ibp://hostname:port/key/WRMKey/WRM
```

There are read, write, and management capabilities with IBP, but with IBPCA there are only read and write capabilities. IBPCA capabilities have this format:

```
Read capabilities: ibpca://hostname:port/R/MD5-checksum
Write capabities: ibpca://hostname:port/W/random_string
```

where

- **hostname** and **port** are the same as in the IBP capability
- **MD5-checksum** is the MD5 hash of the data in the storage depot that the read capability references
- **random_string** is a randomly generated string of length 32

1.4 ibp_probe_info

ibp_probe_info is used in **IBP_CA_manage**.

Variable	Type
size	int
maxSize	int
exists	int
attrib	struct ibp_attributes

When **IBP_CA_manage** is being used with the IBP_CA_MANAGE_PROBE command, the values in the **ibp_probe_info** struct will be filled in with the correct values upon return. Their initial values do not matter.

When **IBP_CA_manage** is being used with the IBP_CA_MANAGE_TIME command, the the duration of the storage will be extended to **attrib.duration** if it is shorter than **attrib.duration**. Nothing happens otherwise. The values of **maxSize**, **exists**, and **attrib** will be updated upon return.

When **IBP_CA_manage** is being used with the IBP_CA_MANAGE_SIZE command, the size of the storage will be increased to **maxSize** if the current size is smaller than **maxSize**. Nothing happens otherwise. The values of **maxSize**, **exists**, and **attrib** will be updated upon return.

When **IBP_CA_manage** is being used with the IBP_CA_MANAGE_DEL command, the values of the **ibp_probe_info** struct are ignored.

2. CLIENT APPLICATIONS

Five client applications are implemented for the IBPCA:

int **IBP_CA_allocate**(IBP_depot depot, IBP_timer timeout, ulong_t maxsize,
IBP_attributes attributes, IBP_cap writecap);

int **IBP_CA_load**(IBP_cap ca_readcap, IBP_timer timeout, char *buf, ulong_t size,
ulong_t offset);

int **IBP_CA_manage**(IBP_cap man_cap, IBP_timer timeout, int cmd, ibp_probe_info
*status);

int **IBP_CA_store**(IBP_cap ca_writecap, IBP_timer timeout, char *data, ulong_t size,
IBP_cap readcap);

int **IBP_CA_store_block**(IBP_depot depot, IBP_timer timeout, ulong_t size, char *data,
IBP_attributes attributes, IBP_cap ca_readcap);

2. 1 IBP_CA_allocate

	variable name	variable type
parameter	depot	IBP_depot
	timeout	IBP_timer
	maxsize	ulong_t
	attributes	IBP_atrributes
(output)	writecap	IBP_cap
Return value		void *

IBP_CA_allocate allocates **maxsize** bytes of storage into the **depot**, with attributes **attributes**. The duration must be finite, and the data type must be byte-array.

Return Values
On successful completion, 0 is returned, otherwise –1 is returned and an error message is sent to stderr. The following conditions will cause IBP_CA_allocate to fail:

- Invalid attributes – For example, infinite duration
- IBP_allocate error – Described in section 3 of IBP v1.1.1 API.

2.2 IBP_CA_load

	variable name	variable type
parameter	readcap	IBP_cap
	timeout	IBP_timer
	buf	char *
	size	ulong_t
Return value		ulong_t

IBP_CA_load loads **size** bytes, starting at the **offset** position, from the byte-array accessed through **readcap** and stores the bytes into **buf**.

Return Values
On successful completion, the number of bytes read is returned, otherwise –1 is returned and an error message is sent to stderr. The following conditions will cause IBP_CA_load to fail:

- Invalid size/offset
- Readcap has an expired duration

- IBPCA internal error
- IBP_load error – Described in section 4 of IBP v1.1.1 API.

2.3 IBP_CA_manage

	variable name	variable type
parameter	man_cap	IBP_cap (a readcap or a writecap)
	timeout	IBP_timer
	cmd	int
	status	ibp_probe_info *
Return value		void

IBP_CA_manage lets the user perform the following operations on a storage area (**IBP_MANAGE_SIZE** and **IBP_CA_MANAGE_DEL** require a writecap):

- **IBP_CA_MANAGE_PROBE** updates **status** with information about the storage area: whether it exists (if the capability is valid), its attributes, size, and maximum capacity.
- **IBP_CA_MANAGE_TIME** extends the duration of a storage area unless the new duration is less than the one that already exists.
- **IBP_CA_MANAGE_SIZE** increases the maximum size of a storage area unless the new size is smaller than the one that already exists.
- **IBP_CA_MANAGE_DEL** deletes a write capability

Return Values
On successful completion, 0 is returned, otherwise –1 is returned and an error message is sent to stderr. The following conditions will cause IBP_CA_manage to fail:

- IBPCA internal error
- Storage area does not exist. That is, writecap/readcap is invalid or expired
- A readcap was sent when a writecap was required
- IBP_manage error – Described in Section 7 of IBP v1.1.1 API.

2.4 IBP_CA_store

	variable name	variable type
parameter	writecap	IBP_cap
	timeout	IBP_timer
	data	char *
	size	ulong_t
(output)	readcap	IBP_cap
Return value		ulong_t

IBP_CA_store appends to a **writecap** previously obtained from **IBP_CA_allocate**. The first **size** bytes of **data** are appended to the byte-array referenced by **writecap**. **Readcap** is then constructed from the entire byte-array's MD5-checksum.

Return values
On successful completion, 0 is returned, otherwise –1 is returned and an error message is sent to stderr. The following conditions will cause IBP_CA_store to fail:

- Expired or Invalid writecap
- IBPCA internal error
- IBP_store error –Described in section 3 of IBP v1.1.1 API.

2.5 IBP_CA_store_block

	variable name	variable type
parameter	depot	IBP_depot
	timeout	IBP_timer
	size	ulong_t
	data	char *
	attributes	IBP_attributes
(output)	readcap	IBP_cap
Return value		int

IBP_CA_store_block allocates and stores **data** into the **depot**. The duration of the storage must be finite, and the data type must be byte-array. The allocated storage space will be read only. If the data to be stored is identical to one already stored, it will not be sent over the network a second time. Instead, the readcap for the original storage will be returned and the duration of the storage will be updated.

Return values
On successful completion, 0 is returned, otherwise –1 is returned and an error message is sent to stderr. The following conditions will cause IBP_CA_store_block to fail:

- Invalid attributes – For example, infinite duration
- IBP_store error – Described in section 3of IBP v1.1.1 API.
- IBP_allocate error – Described in section 2 of IBP v1.1.1 API.

3. INPUT/OUTPUT SUMMARY

IBPCA calls IBP commands to access the IBP depots. The capabilities are treated a bit differently in the IBPCA since there are no manage capabilities and the read capabilities are updated after every store. The following table summarizes the IBP commands called by IBPCA client applications and the role of capabilities in the procedures.

IBPCA **command**	IBP **command**	IBP **Requires**	IBPCA **Requires**	IBP **Returns**	IBPCA **Returns**
IBP_CA_allocate	IBP_allocate			IBP readcap writecap managecap	IBP_CA writecap
IBP_CA_store	IBP_store	IBP writecap	IBP_CA writecap		IBP_CA readcap
IBP_CA_manage	IBP_manage	IBP managecap	IBP_CA writecap **or** readcap		
IBP_CA_store_block	IBP_allocate IBP_store				IBP_CA readcap

4. ADVANTAGES OF USING IBPCA

MD5 is a message-digest algorithm that generates a unique 16 byte string for a data block. This string is used as a checksum in the IBPCA. Changing one bit of the data will change the MD5 checksum that is computed from the data. It is possible to have two files with the same checksum, but it is highly unlikely to happen at random. It is currently computationally infeasible to deliberately create a file with a given checksum or two files with the same checksum [1]. Since there is reasonable assurance that two files will have different checksums, the checksums can be used to distinguish files. The properties of the MD5 hashes give the IBPCA several advantages over the original IBP.

4.1 Checking Integrity of Stored Data

The client can check the integrity of stored data since the MD5 checksum of a file should be the same before and after the file is stored in the IBP depots. The readcap has the original checksum in it, and the checksum of the downloaded data is easily computed.

4.2 Preventing Redundant Network Traffic

When a client wants to store a file, the checksum of the file can be compared to the checksums of files that are already stored. If the file is already stored, then the client just gets access to the existing storage and the data is not transferred a second time. This saves bandwidth and disk space on the depot. IBP_CA_store_block is the only client application that currently implements this feature.

References

[1] Rivest, R. "The MD5 Message-Digest Algorithm", RFC 1321, MIT and RSA Data Security, Inc., April 1992.

```
#include "ibpca_server.h"
#include "ibpca_io.h"
// #include <md5.h>

int receive_ok(int fd) {
  int i;

  i = receive_int(fd);

  if(i != 77) {
    fprintf(stderr, "bad connection\n");
    close(fd);
    pthread_exit(0);
  }

}

/*-----------------------cntl_c_handler--------------------*
 Description: This function cleans up the allocated trees
   when the server is terminated
 *----------------------------------------------------------*/
void cntl_c_handler(int sock)
{
  printf("Done\n");
  clean_up_trees();
  exit(0);
}

/**********************************************************
 *                                                        *
 *                         Main                           *
 *                                                        *
 **********************************************************/
int main(int argc, char **argv)
{
  int sock;
  pthread_t tid;
  pthread_attr_t attr[2];
  void *arg, *arg2;

  signal(SIGINT, cntl_c_handler);

  if(argc != 5 ) {
    fprintf(stderr, "usage: ibpca_server host port ibp_host ibp_port\n");
    exit(1);
  }
  strcpy(ca_depot.ca_host, argv[1]);
  ca_depot.ca_port = atoi(argv[2]);
  if(ca_depot.ca_port < 5000) {
    fprintf(stderr, "Must use a port >= 5000\n");
    exit(1);
  }
  strcpy(ca_depot.host, argv[3]);
  ca_depot.port = atoi(argv[4]);

  sock = serve_known_socket(ca_depot.ca_host, ca_depot.ca_port);
  if(sock < 0) {
    fprintf(stderr, "Error connecting to socket\n");
    exit(1);
  }
  Listen(sock);

  read_cap_dir = make_jrb();
  write_cap_dir = make_jrb();
  expire_list = make_jrb();

  arg = (void *)(&sock);
  pthread_attr_init(attr);
  pthread_attr_setscope(attr, PTHREAD_SCOPE_SYSTEM);

  if(pthread_create(&tid, attr, acceptor, arg) < 0) {
    perror("pthread_create");
    exit(1);
  }
  printf("IBP_CA_server is now ready to accept connections\n");

  pthread_attr_init(attr + 1);
  pthread_attr_setscope(attr + 1, PTHREAD_SCOPE_SYSTEM);
  if(pthread_create(&tid, attr + 1, traverse_expire_list, arg2) < 0) {
    perror("pthread_create");
    exit(1);
  }


  while(1)
    sleep(1000);
}

/*-----------------------acceptor-----------------------*
 Inputs: a socket connection

 Description: Accepts all connections to the server and
  forks off new threads to service each new connection
 *------------------------------------------------------*/
void *acceptor(void *args)
{
  int *fd, sock, *ip;
  pthread_t tid;
  pthread_attr_t attr[2];
  void *arg, *arg2;
  struct sockaddr_in cliaddr;
  int clilen, currenttime;
  JRB tmp;

  ip = (int *)args;
  sock = *ip;

  while(1) {

    clilen = sizeof(cliaddr);
    fd = (int *)malloc(sizeof(int));

    *fd = accept(sock, (struct sockaddr *)&cliaddr, &clilen);
    if(*fd == -1) {
      perror("accept");
      exit(1);
    }
    arg = (void *)(fd);

    pthread_attr_init(attr);
    pthread_attr_setscope(attr, PTHREAD_SCOPE_SYSTEM);

    if(pthread_create(&tid, attr, servicer, arg) < 0) {
      perror("pthread_create");
      exit(1);
    }

  }
  pthread_exit(0);
}

/*--------------------------servicer--------------------------*
 Inputs: a file descriptor to the connection

 Description: services a connection
 *------------------------------------------------------------*/
```

```c
void *servicer(void *args)
{
  int fd, *ip;

  ip = (int *)args;
  fd = *ip;
  free(ip);

//   receive_ok(fd);

  handle_ca_request(fd);

  close(fd);
  pthread_exit(0);
}


/*---------------------handle_ca_request-----------------------*
 Inputs: a file descriptor

 Description: First receives the command packet, and then calls
  the appropriate function to handle the command
 *-------------------------------------------------------------*/
void handle_ca_request(int fd)
{
  ibp_ca_command ibp_ca_cmd;

  decode_cmd_ca(fd, &ibp_ca_cmd);

  switch(ibp_ca_cmd.Cmd) {
    case IBP_CA_STORE:
      printf("IBP_CA_STORE\n");
      handle_ca_store(fd, ibp_ca_cmd);
      break;
    case IBP_CA_ALLOCATE:
      printf("IBP_CA_ALLOCATE\n");
      handle_ca_allocate(fd, ibp_ca_cmd);
      break;
    case IBP_CA_STORE_BLOCK:
      printf("IBP_CA_STORE_BLOCK\n");
      handle_ca_store_block(fd, ibp_ca_cmd);
      break;
    case IBP_CA_LOAD:
      printf("IBP_CA_LOAD\n");
      handle_ca_load(fd, ibp_ca_cmd);
      break;
    case IBP_CA_MANAGE:
      printf("IBP_CA_MANAGE\n");
      handle_ca_manage(fd, ibp_ca_cmd);
      break;
  }
}


/*-----------------------decode_cmd_ca----------------------*
 Inputs:
  a file descriptor, and command information

 Description:
  Receives a packet of type IBP_CA_packet from the client and
  transfers the information to a ibp_ca_command struct for
  further use
 *----------------------------------------------------------*/
void *decode_cmd_ca(int fd, ibp_ca_command *ibp_ca_cmd)
{
  char *buf, *cp;
  long bytes_read, bytes_remaining;
  IBP_CA_packet packet;

  buf = (char *)malloc(CMD_BUF_SIZE*sizeof(char));

  bytes_read = read(fd, (void *)&packet, sizeof(packet));

  if(bytes_read != sizeof(packet)) {

    bytes_remaining = sizeof(packet) - bytes_read;
    bytes_read = read(fd, (void *)&packet + bytes_read, bytes_remaining);

    if(bytes_read != bytes_remaining) {
    fprintf(stderr, "bad packet size %d, fd = %d\n",
                    sizeof(packet) + bytes_read + bytes_remaining, fd);
      exit(1);
    }
  }

  ibp_ca_cmd->Cmd = ntohl(packet.Cmd);
  ibp_ca_cmd->timeout.ServerSync = ntohl(packet.ServerSync);
  ibp_ca_cmd->timeout.ClientTimeout = ntohl(packet.ClientTimeout);
  ibp_ca_cmd->attributes.reliability = ntohl(packet.reliability);
  ibp_ca_cmd->attributes.duration = ntohl(packet.duration);
  ibp_ca_cmd->attributes.type = IBP_BYTEARRAY;
  ibp_ca_cmd->size = ntohl(packet.size);
  ibp_ca_cmd->lifetime = ntohl(packet.lifetime);
  ibp_ca_cmd->offset = ntohl(packet.offset);
  ibp_ca_cmd->manage_cmd = ntohl(packet.man_cmd);
  if(strcmp(packet.cap, "") != 0) {
    ibp_ca_cmd->cap = (char *)malloc(IBPCA_CAP_LEN);
    strncpy(ibp_ca_cmd->cap, packet.cap, IBPCA_CAP_LEN);
  }
  strncpy(ibp_ca_cmd->checksum, packet.checksum, CHECKSUM_SIZE);

  if(ibp_ca_cmd->Cmd ==  IBP_CA_MANAGE) {

    if((cp = strstr(ibp_ca_cmd->cap, "/R/")) != NULL) {
      ibp_ca_cmd->man_ctype = 'r';
      strcpy(ibp_ca_cmd->checksum, (char *)(cp + 3));
    } else if((cp = strstr(ibp_ca_cmd->cap, "/W/")) != NULL) {
      ibp_ca_cmd->man_ctype = 'w';
//      strcpy(ibp_ca_cmd->checksum, (char *)(cp + 3));
    }
  }
}


/*-----------------------------------------------------------*

      Handles the client request for IBP_CA_store

 *-----------------------------------------------------------*/
void handle_ca_store(int fd, ibp_ca_command ibp_ca_cmd)
{
  JRB tmp;
  int ok, duration;
  IBP_set_of_caps socaps;
  char *oldchecksum, *checksum, *readcap;
  char oldreadcap[IBPCA_CAP_LEN], output[HASH_SIZE];
  char ibp_read[IBP_CAP_LEN], ibp_write[IBP_CAP_LEN], ibp_manage[IBP_CAP_LEN];
  read_info *rinfo;
  time_t tloc;
  write_info *winfo;
  MD5_CTX context, new_context;

  // figure out ibp write_cap from ibp_ca write_cap

  pthread_mutex_lock(&write_lock);
  tmp = jrb_find_str(write_cap_dir, ibp_ca_cmd.cap);
```

```
  if(tmp == NULL) {
    fprintf(stderr, "Invalid writecap\n");
    send_int(fd, -1);
    pthread_mutex_unlock(&write_lock);
    return;
  }

  winfo = (write_info *)jval_v(tmp->val);
  context = winfo->context;
  duration = winfo->duration;
  strncpy(ibp_read, winfo->socaps->readCap, IBP_CAP_LEN);
  strncpy(ibp_write, winfo->socaps->writeCap, IBP_CAP_LEN);
  strncpy(ibp_manage, winfo->socaps->manageCap, IBP_CAP_LEN);

  // check to see that the duration has not expired
  time(&tloc);
  if(winfo->duration < tloc) {
    pthread_mutex_unlock(&write_lock);
    fprintf(stderr, "duration of %s has expired\n", ibp_ca_cmd.cap);
    send_int(fd, -2);
    return;
  }
  send_int(fd, 1);
  ibp_ca_cmd.attributes.duration = duration;

  // tell client to store the data
  send_string(ibp_write, fd);
  send_context(context, fd);

  ok = receive_int(fd);
  if(ok == 0) {
    pthread_mutex_unlock(&write_lock);
    fprintf(stderr, "store didn't work\n");
    return;
  }
  new_context = receive_context(fd);

  winfo->context = new_context;
  winfo->currentsize += ok;
  pthread_mutex_unlock(&write_lock);

//  output = (char *)malloc(17);

  MD5Final(output, &context);

  oldchecksum = (char *)hash2hex(output);
  construct_readcap(oldchecksum, oldreadcap);

  MD5Final(output, &new_context);

  checksum = (char *)hash2hex(output);

  // insert it into tree, return read-cap
  pthread_mutex_lock(&read_lock);
  tmp = jrb_find_str(read_cap_dir, oldreadcap);

  if(tmp != NULL) {
    rinfo = (read_info *)jval_v(tmp->val);

    ibp_ca_cmd.attributes.duration = rinfo->duration;
    ibp_ca_cmd.lifetime = rinfo->lifetime;

    ibp_ca_cmd.size += rinfo->size;

    pthread_mutex_unlock(&read_lock);
    readcap = AddtoReadCapDir(checksum, ibp_ca_cmd, ibp_manage, ibp_read);
    if(readcap == NULL) {
      fprintf(stderr, "error adding new checksum to checksum directory\n");
      send_string("", fd);
      return;
    }
  } else {
    pthread_mutex_unlock(&read_lock);

//    construct_readcap(checksum, readcap);
    readcap = AddtoReadCapDir(checksum, ibp_ca_cmd, ibp_manage, ibp_read);
  }

//  printf("sending readcap %s\n", readcap);
  send_string(readcap, fd);

  free(readcap);
}

/*---------------------------------------------------------------------*

       Handles the client request for IBP_CA_allocate

 *---------------------------------------------------------------------*/
void handle_ca_allocate(int fd, ibp_ca_command ibp_ca_cmd)
{
  IBP_set_of_caps socaps;
  struct ibp_depot depot;
  struct ibp_timer timeout;
  IBP_CA_cap ca_writecap;
  char *random_str;

  // initialize depot and attributes
  strcpy(depot.host, ca_depot.host);
  depot.port = ca_depot.port;

  socaps = (IBP_set_of_caps)malloc(sizeof(struct ibp_set_of_caps));
  socaps = (IBP_set_of_caps)IBP_allocate(&depot, &(ibp_ca_cmd.timeout),
            ibp_ca_cmd.size, ibp_ca_cmd.attributes);
  if(socaps == NULL) {
    fprintf(stderr, "error with IBP_allocate: %d\n", IBP_errno);
    send_int(fd, IBP_errno);
    return;
  }
  send_int(fd, IBP_OK);

  // create ibp_ca write cap add to write_cap_dir
  random_str = tmpnam(NULL);
  md5_calc(random_str, random_str, strlen(random_str));
  random_str = (char *)hash2hex(random_str);

  ca_writecap = (char *)malloc(IBPCA_CAP_LEN);

  sprintf(ca_writecap, "ibpca://%s:%d/W/%s", ca_depot.ca_host,
          ca_depot.ca_port, random_str);
  AddtoWriteCapDir(ca_writecap, socaps, ibp_ca_cmd);

  // return writecap to client
  send_string(ca_writecap, fd);
}

/*---------------------------------------------------------------------*

       Handles the client request for IBP_CA_store_block

 *---------------------------------------------------------------------*/
void handle_ca_store_block(int fd, ibp_ca_command ibp_ca_cmd)
{
  char capability[IBPCA_CAP_LEN], ibp_manage[IBP_CAP_LEN];
  JRB tmp, tmp2;
```

```c
  IBP_set_of_caps socaps;
  struct ibp_depot depot;
  ulong_t length;
  time_t expireTime;
  read_info *rinfo;
  struct ibp_capstatus info;
  int duration;

  if(ibp_ca_cmd.timeout.ServerSync != 0)
    expireTime = time(NULL) + ibp_ca_cmd.timeout.ServerSync;
  else
    expireTime = 0;

  construct_readcap(ibp_ca_cmd.checksum, capability);

  // check to see if checksum already exists
  pthread_mutex_lock(&read_lock);
  tmp = jrb_find_str(read_cap_dir, capability);
  if(tmp != NULL) {
     rinfo = (read_info *)jval_v(tmp->val);
     if(rinfo->duration < time(0)) tmp = NULL;
  }
  if(tmp == NULL) {
    pthread_mutex_unlock(&read_lock);
    printf("checksum not found\n");
    send_int(fd, 0);

    // attempt to allocate and store data
    strcpy(depot.host, ca_depot.host);
    depot.port = ca_depot.port;

    socaps = (IBP_set_of_caps)malloc(sizeof(struct ibp_set_of_caps));
    socaps = (IBP_set_of_caps)IBP_allocate(&depot,
                         &(ibp_ca_cmd.timeout),
                         ibp_ca_cmd.size,
                         &(ibp_ca_cmd.attributes));

    if(socaps == NULL) {
      fprintf(stderr, "IBP_allocate: error code %d\n", IBP_errno);
      send_int(fd, IBP_errno);
      return;
    }
    send_int(fd, IBP_OK);

    send_string(socaps->writeCap, fd);
    length = receive_int(fd);

    ibp_ca_cmd.cap = strdup((char *)socaps->writeCap);

//    AddtoWriteCapDir(socaps->writeCap, socaps, ibp_ca_cmd);
    AddtoReadCapDir(ibp_ca_cmd.checksum, ibp_ca_cmd, socaps->manageCap, socaps->readCa
p);

  } else {
    // get management cap for checksum
    rinfo = (read_info *)jval_v(tmp->val);
    strncpy(ibp_manage, rinfo->manageCap, IBP_CAP_LEN);
    duration = rinfo->duration;
    pthread_mutex_unlock(&read_lock);

    printf("checksum found\n");
    send_int(fd, 1);

    if(duration <  ibp_ca_cmd.attributes.duration) {
      if(IBP_manage(ibp_manage, &(ibp_ca_cmd.timeout),
                    IBP_PROBE, IBP_READCAP, &info) != 0) {
        fprintf(stderr, "ibp_manage error while probing %d\n", IBP_errno);
        printf("duration = %d\n", duration);
        printf("time now = %d\n", time(0));
        send_int(fd, 2);
      }
      info.attrib.duration = ibp_ca_cmd.attributes.duration;

      if(IBP_manage(ibp_manage, &(ibp_ca_cmd.timeout),
                    IBP_CHNG, IBP_WRITECAP, &info) != 0) {
        fprintf(stderr, "error with IBP_manage %d\n", IBP_errno);
        send_int(fd, 2);
      }
      else send_int(fd, 0);

      // need to pthread lock?
      rinfo->duration = info.attrib.duration;

    } else send_int(fd, 0);
  }

  send_string(capability, fd);
  return;

}

/*------------------------AddtoReadCapDir--------------------*
 * Description: Adds the new readcap to the readcap directory   *
 *-----------------------------------------------------------*/
char *AddtoReadCapDir(char *checksum, ibp_ca_command ibp_ca_cmd,
                      IBP_cap manageCap, IBP_CA_cap readCap)
{
  read_info *rinfo;
  char *capability;
  JRB tmp;

  rinfo = (read_info *)malloc(sizeof(read_info));
  rinfo->duration = ibp_ca_cmd.attributes.duration;
  rinfo->lifetime = ibp_ca_cmd.lifetime;
  rinfo->size = ibp_ca_cmd.size;
  rinfo->manageCap = strdup(manageCap);
  rinfo->readCap = strdup(readCap);

  capability = (char *)malloc(IBPCA_CAP_LEN);
  construct_readcap(checksum, capability);

  pthread_mutex_lock(&read_lock);
  tmp = jrb_insert_str(read_cap_dir, strdup(capability),
                       new_jval_v((void *)rinfo));
  if(tmp == NULL) {
    perror("jrb_insert_str");
    exit(1);
  }

  tmp = addto_expire_list(tmp, rinfo->duration, capability);
  if(tmp == NULL) {
    fprintf(stderr, "IBPCA internal error\n");
    exit(0);   // or something
  }
  rinfo->exp_nod = tmp;
  pthread_mutex_unlock(&read_lock);

  return(capability);
}

/*-------------------AddtoWriteCapDir----------------------*
 * Description: Adds the new writecap to the writecap         *
 *       directory                                         *
 *---------------------------------------------------------*/
void AddtoWriteCapDir(IBP_CA_cap writecap, IBP_set_of_caps socaps,
                      ibp_ca_command ibp_ca_cmd)
{
```

```
  JRB tmp;
  write_info *winfo;

  winfo = (write_info *)malloc(sizeof(write_info));
  winfo->socaps = socaps;
  winfo->maxsize = ibp_ca_cmd.size;
  winfo->currentsize = 0;
  winfo->duration = ibp_ca_cmd.attributes.duration;
  MD5Init(&(winfo->context));

  pthread_mutex_lock(&write_lock);
  tmp = jrb_insert_str(write_cap_dir, strdup((char *)writecap),
                       new_jval_v((void *)winfo) );

  if(ibp_ca_cmd.Cmd != IBP_CA_STORE_BLOCK) {
    tmp = addto_expire_list(tmp, winfo->duration, writecap);
    if(tmp == NULL) {
      fprintf(stderr, "internal error");
      exit(0);
    }
    winfo->exp_nod = tmp;
  }
  pthread_mutex_unlock(&write_lock);

}

/*------------------------------------------------------------------*

       Handles the client request for IBP_CA_manage

 *------------------------------------------------------------------*/
void handle_ca_manage(int fd, ibp_ca_command ibp_ca_cmd)
{
  char *checksum, ibp_manage[IBP_CAP_LEN];
  JRB tmp;
  int duration;
  ulong_t size;
  read_info *rinfo;
  IBP_set_of_caps socaps;
  time_t tloc;
  write_info *winfo;

  if(ibp_ca_cmd.man_ctype == 'r' &&
     (ibp_ca_cmd.manage_cmd == IBP_CA_MANAGE_SIZE ||
      ibp_ca_cmd.manage_cmd == IBP_CA_MANAGE_DEL)) {
    fprintf(stderr, "Must have write-cap to change size or delete a
                      write-cap\n");
    send_int(fd, NEED_WRITE_CAP);
    return;
  }

  // figure out ibp manage cap if write_cap, identification if read_cap
  if(ibp_ca_cmd.man_ctype == 'r') {
    checksum = extract_checksum(ibp_ca_cmd.cap);

    pthread_mutex_lock(&read_lock);
    tmp = jrb_find_str(read_cap_dir, ibp_ca_cmd.cap);

    if(tmp == NULL) {
      pthread_mutex_unlock(&read_lock);
      fprintf(stderr, "didn't find read-cap %s\n", ibp_ca_cmd.cap);
      send_int(fd, BAD_READ_WRITE_CAP);
      return;
    }
    rinfo = (read_info *)jval_v(tmp->val);
    strncpy(ibp_manage, rinfo->manageCap, IBP_CAP_LEN);
    duration = rinfo->duration;
    size = rinfo->size;
```

```
    pthread_mutex_unlock(&read_lock);

  } else {

    pthread_mutex_lock(&write_lock);
    tmp = jrb_find_str(write_cap_dir, ibp_ca_cmd.cap);

    if(tmp == NULL) {
      pthread_mutex_unlock(&write_lock);
      fprintf(stderr, "didn't find write_cap %s\n", ibp_ca_cmd.cap);
      send_int(fd, BAD_READ_WRITE_CAP);
      return;
    }
    winfo = (write_info *)jval_v(tmp->val);
    strncpy(ibp_manage, winfo->socaps->manageCap, IBP_CAP_LEN);
    duration = winfo->duration;
    size = winfo->maxsize;
    pthread_mutex_unlock(&write_lock);
  }
  time(&tloc);
  if(duration < tloc) {
    fprintf(stderr, "duration of %s has expired\n", ibp_ca_cmd.cap);
    send_int(fd, DURATION_EXPIRED);
    return;
  }

  switch(ibp_ca_cmd.manage_cmd) {
    case IBP_CA_MANAGE_PROBE:
      handle_ca_manage_probe(ibp_ca_cmd, ibp_manage, size, duration, fd);
      break;
    case IBP_CA_MANAGE_TIME:
      handle_ca_manage_time(ibp_ca_cmd, ibp_manage, size, fd);
      break;
    case IBP_CA_MANAGE_SIZE:
      handle_ca_manage_size(ibp_ca_cmd, ibp_manage, duration, fd);
      break;
    case IBP_CA_MANAGE_DEL:
      handle_ca_manage_del(ibp_ca_cmd, ibp_manage, fd);
      break;
  }
}

/*-------------------handle_ca_manage_probe-------------------*

 Description: handles client requests to probe a read or
              writecap

 *-----------------------------------------------------------*/
void handle_ca_manage_probe(ibp_ca_command ibp_ca_cmd, char *manageCap,
                            int size, int duration, int fd)
{
  struct ibp_capstatus info;

  printf("       MANAGE_PROBE\n");
  if(ibp_ca_cmd.man_ctype == 'w') {
    send_int(fd, size);
    send_int(fd, duration);
  } else if(ibp_ca_cmd.man_ctype == 'r') {
    send_int(fd, size);
    send_int(fd, duration);
  }

}

/*-------------------handle_ca_manage_time---------------------*

 Description: handles client requests to extend the duration of
              a writecap
```

```
 *--------------------------------------------------------------*/
void handle_ca_manage_time(ibp_ca_command ibp_ca_cmd, char *manageCap,
                           int size, int fd)
{
  struct ibp_capstatus info;

  printf("       MANAGE_TIME\n");

  send_int(fd, IBPCA_OK);
  info.attrib.duration = ibp_ca_cmd.attributes.duration;
  info.attrib.type = IBP_BYTEARRAY;
  info.attrib.reliability = IBP_STABLE;
  info.maxSize = size;

  if(IBP_manage(manageCap, &(ibp_ca_cmd.timeout),
                IBP_CHNG, IBP_READCAP, &info) != 0) {
    fprintf(stderr, "error with ibp_manage\n");
    send_int(fd, IBP_errno);
    return;
  }
  send_int(fd, IBPCA_OK);

  update_expire_list(ibp_ca_cmd.attributes.duration, ibp_ca_cmd.cap);

}

/*------------------handle_ca_manage_size-----------------------*

 Description: handles client requests to increase the size of
              the storage area accesed by a writecap

 *--------------------------------------------------------------*/
void handle_ca_manage_size(ibp_ca_command ibp_ca_cmd, char *manageCap,
                           int duration, int fd)
{
  struct ibp_capstatus info;
  JRB tmp;
  write_info *winfo;

  printf("       MANAGE_SIZE\n");

  send_int(fd, IBPCA_OK);

  info.attrib.duration = duration;
  info.attrib.type = IBP_BYTEARRAY;
  info.attrib.reliability = IBP_STABLE;
  info.maxSize = ibp_ca_cmd.size;

  if( IBP_manage(manageCap, &(ibp_ca_cmd.timeout),
                 IBP_CHNG, IBP_READCAP, &info) != 0) {
    fprintf(stderr, "error with ibp_manage %d\n", IBP_errno);
    send_int(fd, IBP_errno);
    return;
  }

  // update writecap maxsize
  pthread_mutex_lock(&write_lock);
  tmp = jrb_find_str(write_cap_dir, ibp_ca_cmd.cap);
  winfo = (write_info *)jval_v(tmp->val);
  winfo->maxsize = ibp_ca_cmd.size;
  pthread_mutex_unlock(&write_lock);

  send_int(fd, IBPCA_OK);
}


/*------------------handle_ca_manage_del------------------------*

 Description: handles client requests to delete a writecap

 *--------------------------------------------------------------*/
void handle_ca_manage_del(ibp_ca_command ibp_ca_cmd, char *manageCap, int fd)
{
  JRB tmp, dur_tmp;
  write_info *winfo;
  struct ibp_capstatus info;

  printf("       MANAGE_DEL\n");

  // delete from IBP
/*
  if(IBP_manage(manageCap, &(ibp_ca_cmd.timeout), IBP_DECR,
                IBP_READCAP, &info) != 0) {
    fprintf(stderr, "ibp_manage error while probing %d\n", IBP_errno);
    send_int(fd, IBP_errno);
    return;
  }
*/

  // delete from IBPCA
  pthread_mutex_lock(&expire_lock);
  pthread_mutex_lock(&write_lock);

  tmp = jrb_find_str(write_cap_dir, ibp_ca_cmd.cap);
  winfo = (write_info *)jval_v(tmp->val);

  dur_tmp = winfo->exp_nod;
  pthread_mutex_unlock(&write_lock);

  remove_expire_node(dur_tmp);
  pthread_mutex_unlock(&expire_lock);

  send_int(fd, IBPCA_OK);
  send_int(fd, IBPCA_OK);

}

/*----------------------delete_write_cap----------------------*
 Description: Deletes a writecap from the writecap directory
    !! This function is probably unnecesary now -- probably
    can rewrite clean_up_trees with remove_expire_node
 *------------------------------------------------------------*/
void delete_write_cap(JRB tmp)
{
   IBP_set_of_caps socaps;
   write_info *winfo;
   char *ch;

  if(tmp != NULL) {
    winfo = (write_info *)jval_v(tmp->val);

    if(winfo != NULL ) {

      socaps = winfo->socaps;
      if(socaps != NULL) {
        if(socaps >readCap != NULL) free(socaps->readCap);
        if(socaps->writeCap != NULL) free(socaps->writeCap);
        if(socaps->manageCap != NULL) free(socaps->manageCap);
        free(socaps);
      }
      free(winfo);

    }
    ch = (char *)jval_s(tmp->key);
    free(ch);
```

```
    pthread_mutex_lock(&write_lock);
    jrb_delete_node(tmp);
    pthread_mutex_unlock(&write_lock);
  }
}

/*-------------------------------------------------------------*

        Handles the client request for IBP_CA_load

 *-------------------------------------------------------------*/
void handle_ca_load(int fd, ibp_ca_command ibp_ca_cmd)
{
  JRB tmp;
  char *checksum, ibp_readcap[IBP_CAP_LEN], *ch;
  read_info *rinfo;
  time_t tloc;
  int size, duration;

  checksum = extract_checksum(ibp_ca_cmd.cap);

  pthread_mutex_lock(&read_lock);
  tmp = jrb_find_str(read_cap_dir, ibp_ca_cmd.cap);

  if(tmp == NULL) {
    pthread_mutex_unlock(&read_lock);
    fprintf(stderr, "Invalid checksum\n");
    return;
  }
  rinfo = (read_info *)jval_v(tmp->val);
  size = rinfo->size;
  duration = rinfo->duration;
  strncpy(ibp_readcap, rinfo->readCap, IBP_CAP_LEN);
  pthread_mutex_unlock(&read_lock);

  time(&tloc);
  if(duration < tloc) {
    fprintf(stderr, "duration of %s has expired\n", ibp_ca_cmd.cap);
    send_int(fd, DURATION_EXPIRED);
    return;
  }
  send_int(fd, IBPCA_OK);

  send_int(fd, size);
  send_string(ibp_readcap, fd);
}

/*---------------------clean_up_trees----------------------*
  Description: frees up all of the memory from the global
        trees
 *-------------------------------------------------------------*/
void clean_up_trees()
{
  JRB tmp, dur_tmp;
  read_info *rinfo;
  write_info *winfo;
  char *ch;

  pthread_mutex_lock(&expire_lock);

  jrb_traverse(tmp, write_cap_dir) {
    winfo = (write_info *)jval_v(tmp->val);
    dur_tmp = winfo->exp_nod;
    remove_expire_node(dur_tmp);
  }
  jrb_free_tree(write_cap_dir);

  jrb_traverse(tmp, read_cap_dir) {
    rinfo = (read_info *)jval_v(tmp->val);
    dur_tmp = rinfo->exp_nod;
    remove_expire_node(dur_tmp);
  }
  jrb_free_tree(read_cap_dir);

  pthread_mutex_unlock(&expire_lock);

}

/*--------------------addto_expire_list--------------------*
 Input: new JRB node from readcap or writecap directory,
        duration and cap of new node
 Description: Adds the new node and its duration to the
        expire list
 *-------------------------------------------------------*/
JRB addto_expire_list(JRB new, int duration, char *cap)
{
  JRB tmp;

  pthread_mutex_lock(&expire_lock);
  tmp = jrb_insert_int(expire_list, duration, new_jval_v((void *)new));
  pthread_mutex_unlock(&expire_lock);

  return(tmp);
}

/*---------------------update_expire_list------------------*
 Description: updates the duration of a read or writecap in
        the expire list
 *-------------------------------------------------------*/
void update_expire_list(int new_dur, char *cap)
{
  JRB tmp, update, old_dur;
  char type;
  write_info *winfo;
  read_info *rinfo;

  if(strstr(cap, "/R/") == NULL) {
    type = 'w';
    pthread_mutex_lock(&write_lock);
  }
  else {
    type = 'r';
    pthread_mutex_lock(&read_lock);
  }

  pthread_mutex_lock(&expire_lock);

  if(type == 'w') {
    tmp = jrb_find_str(write_cap_dir, cap);
    winfo = (write_info *)jval_v(tmp->val);
    old_dur = winfo->exp_nod;
  } else {
    tmp = jrb_find_str(read_cap_dir, cap);
    rinfo = (read_info *)jval_v(tmp->val);
    old_dur = rinfo->exp_nod;
  }

  jrb_delete_node(old_dur);
  update = jrb_insert_int(expire_list, new_dur, new_jval_v((void *)tmp));

  if(type == 'w') {
    winfo->exp_nod = update;
    winfo->duration = new_dur;
    pthread_mutex_unlock(&write_lock);
  } else {
    rinfo->exp_nod = update;
```

```
    rinfo->duration = new_dur;
    pthread_mutex_unlock(&read_lock);
  }

  pthread_mutex_unlock(&expire_lock);
}

/*-----------------traverse_expire_list---------------------*
 Description: traverses the expire list and deletes all nodes
       whose durations have expired
 *----------------------------------------------------------*/
void *traverse_expire_list(void *arg)
{
  JRB dur_tmp, tmp;
  int expire_dur, next_expire, mpty;

  while(1) {

    printf("TRAVERSE_EXPIRE_LIST\n");

    expire_dur = time(0);
    mpty = 1;

    pthread_mutex_lock(&expire_lock);
    jrb_traverse(dur_tmp, expire_list){
      if(jval_i(dur_tmp->key) < expire_dur){
        tmp = (JRB)jval_v(dur_tmp->val);
        remove_expire_node(dur_tmp);
      } else {
        mpty = 0;
        next_expire = jval_i(dur_tmp->key);
        break;
      }
    }
    pthread_mutex_unlock(&expire_lock);

    if(!mpty) next_expire = next_expire - time(0);
    if(mpty || next_expire == 0 || next_expire > 25)
      next_expire = 25;

    printf("about to sleep for %d seconds\n", next_expire);
    sleep(next_expire);

  }

}

/*-----------------remove_expire_node-----------------------*
 Description: Removes a capability from its directory and the
       expire list.  Whenever this function is called, the
       expire_lock is mutex locked from the outside.

 *----------------------------------------------------------*/
void remove_expire_node(JRB dur_tmp)
{
  JRB cap_tmp, check_tmp;
  char cap[IBPCA_CAP_LEN], *checksum, *cp;
  read_info *rinfo;
  write_info *winfo;

  cap_tmp = (JRB)jval_v(dur_tmp->val);
  strcpy(cap, jval_s(cap_tmp->key));

  printf("deleting cap %s\n", cap);

  cp = strstr(cap, "/R/");
  if(cp != NULL) {

    // delete node from read_cap_dir

    pthread_mutex_lock(&read_lock);

    rinfo = (read_info *)jval_v(cap_tmp->val);
    free(rinfo->manageCap);
    free(rinfo->readCap);
    free(rinfo);

    jrb_delete_node(cap_tmp);
    pthread_mutex_unlock(&read_lock);

  } else {

    // delete node from writecap_dir
    pthread_mutex_lock(&write_lock);
    winfo = (write_info *)jval_v(cap_tmp->val);
    free(winfo->socaps->readCap);
    free(winfo->socaps->writeCap);
    free(winfo->socaps->manageCap);
    free(winfo);

    jrb_delete_node(cap_tmp);
    pthread_mutex_unlock(&write_lock);

  }

  jrb_delete_node(dur_tmp);
}

/*----------------------------------------------------------------------------*
  Description: extracts a checksum from a readcap;

  Returns: the checksum if the readcap is valid, otherwise NULL is returned
 *----------------------------------------------------------------------------*/
char *extract_checksum(char *readcap)
{
  char *checksum, *tmp;

  checksum = (char *)malloc(CHECKSUM_SIZE);

  tmp = strstr(readcap, "/R/");
  if(tmp == NULL) return NULL;

  strncpy(checksum, tmp+3, CHECKSUM_SIZE);

  return checksum;
}

/*----------------------------------------------------------------------------*
  Description: constructs a readcap from a checksum

  Returns: the readcap
 *----------------------------------------------------------------------------*/
void construct_readcap(char *checksum, char *readcap)
{
  sprintf(readcap, "ibpca://%s:%d/R/%s", ca_depot.ca_host, ca_depot.ca_port, checksum)
;
}
```

```c
#include <errno.h>
#include <math.h>
// #include </usr/include/md5.h>
#include <md5.h>
#include <pthread.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <time.h>
#include <unistd.h>
#include "ibp_ca.h"
#include "ibp_errno.h"
#include "ibp_server.h"
#include "ibp_protocol.h"
#include "jrb.h"
#include "libend2end.h"

typedef struct {
  int Cmd;
  int lifetime;
  int port;
  int manage_cmd;
  char man_ctype;
  char *host;
  char checksum[33];
  IBP_cap cap;
  unsigned long size;
  unsigned long offset;
  struct ibp_attributes attributes;
  struct ibp_timer timeout;
} ibp_ca_command;

typedef struct {
  int duration;
  int reliability;
  int lifetime;
  ulong_t size;
  IBP_cap manageCap;
  IBP_cap readCap;
  JRB exp_nod;
} read_info;

typedef struct {
  IBP_set_of_caps socaps;
  int maxsize;
  int currentsize;
  int duration;
  MD5_CTX context;
  JRB exp_nod;
} write_info;

void *acceptor(void *);
void *servicer(void *);
void *decode_cmd_ca(int, ibp_ca_command *);
void handle_ca_request(int);
void handle_ca_store_block(int, ibp_ca_command);
void handle_ca_allocate(int, ibp_ca_command);
void handle_ca_store(int, ibp_ca_command);
void handle_ca_load(int, ibp_ca_command);
void handle_ca_manage(int, ibp_ca_command);
void handle_ca_manage_probe(ibp_ca_command, char *, int, int, int);
void handle_ca_manage_time(ibp_ca_command, char *, int, int);
void handle_ca_manage_size(ibp_ca_command, char *, int, int);
void handle_ca_manage_del(ibp_ca_command, char *, int);
char *AddtoReadCapDir(char *, ibp_ca_command, IBP_cap, IBP_CA_cap);
void AddtoWriteCapDir(IBP_CA_cap, IBP_set_of_caps, ibp_ca_command);
JRB addto_expire_list(JRB, int, char *);
void send_int(int, int);
int receive_int(int);
void send_context(MD5_CTX, int);
MD5_CTX receive_context(int);
void *traverse_expire_list(void *);
void update_expire_list(int, char *);
void remove_expire_node(JRB);
void delete_write_cap(JRB);
void clean_up_trees();
char *extract_checksum(char *);
void construct_readcap(char *, char *);

/* Globals */
pthread_mutex_t write_lock;
pthread_mutex_t read_lock;
pthread_mutex_t expire_lock;
pthread_mutex_t send_lock;

JRB read_cap_dir;
JRB write_cap_dir;
JRB expire_list;
// JRB expirec_list;  !! needs to be deleted in src and incorporated into
//                          read and write dirs
```

```c
#include "ibp_ca.h"
#include <sys/types.h>
#include <sys/socket.h>
#include <stdlib.h>
#include <unistd.h>
#include <strings.h>
#include <sys/time.h>
#include <fcntl.h>
#include <netinet/in.h>
#include <netdb.h>
#include "ibpca_io.h"

char *CalculateChecksum(unsigned char *ablock, int size);
int create_ca_IURL(char *cap, IBP_timer);
void send_packet(int, int, int, int, int, int, int, int, int,
                 int, char *, char *);
int ca_connect_socket(char *, int, int);

void send_ok(int fd) {

  send_int(77, fd);

}


/*------------------IBP_CA_store_block-------------------*
  Description:
  IBP_CA_store_block allocates and stores data into the
  depot.  The duration of the storage must be finite,
  and the data type must be byte-array.  The allocated
  storage space will be read only.  If the data to be
  stored is identical to one already stored, it will
  not be stored a second time.  Instead, the readcap
  for the original storage will be returned and the
  duration of the storage will be updated.

  Returns:
  0 on successful completion, -1 otherwise
 *-------------------------------------------------------*/
int IBP_CA_store_block(      IBP_depot depot,
                             IBP_timer timeout,
                             ulong_t size,
                             char *data,
                             IBP_attributes attributes,
                             IBP_CA_cap ca_readcap)
{
  int fd, length, exists, ok;
  time_t lt_now, lt_lifetime;
  char *checksum, s[33];
  char lc_buffer[CMD_BUF_SIZE];
  IBP_cap writecap;

  checksum = (char *)malloc(33*sizeof(char));
  checksum = CalculateChecksum(data, size);

  /*
   * check IBP attributes
   */
  time(&lt_now);
  if ( attributes == NULL ) {
    fprintf(stderr, "error: invalid attributes\n");
    return(-1);
  }
  if ( attributes->duration > 0) {
    lt_lifetime = attributes->duration - lt_now;
    if ( lt_lifetime < 0){
      fprintf(stderr, "error: can't have negative lifespan\n");
      return(-1);
    }
  } else {
    fprintf(stderr, "duration must be finite\n");
    return(-1);
  }

  if (d_CheckAttrRely(attributes->reliability) != IBP_OK) {
    fprintf(stderr, "error: invalid reliability\n");
    return(-1);
  }

  if(attributes->type != IBP_BYTEARRAY) {
    fprintf(stderr, "Data type must be bytearray\n");
    return(-1);
  }

  /*
   * connect to IBP server
   */
//  fd = request_connection(depot->host, depot->port);

  fd = ca_connect_socket(depot->host, depot->port, timeout->ClientTimeout);
  if(fd == -1) {
    fprintf(stderr, "Connection to %s:%d failed\n", depot->host, depot->port);
    return(-1);
  }

//  send_ok(fd);

  send_packet(fd, IBP_CA_STORE_BLOCK, timeout->ServerSync,
              timeout->ClientTimeout, attributes->reliability,
              attributes->duration, size, lt_lifetime, 0, 0,
              checksum, NULL);

  exists = receive_int(fd);

  if(!exists) {
    ok = receive_int(fd);
    if(ok != IBP_OK) {
      fprintf(stderr, "IBP_allocate: %d\n", ok);
      close(fd);
      return(-1);
    }

    writecap = (IBP_cap)malloc(IBP_CAP_LEN);
    receive_string(writecap, IBP_CAP_LEN, fd);

// printf("writecap is %s\n", writecap);

    ok = IBP_store(writecap, &timeout, data, size);
    if(ok == 0) {
      fprintf(stderr, "IBP_store: %d\n", IBP_errno);
      close(fd);
      return(-1);
    }
    send_int(fd, ok);

  } else {
    ok = receive_int(fd);
    if(ok == 2) {
      printf("couldn\'t modify duration\n");
    }
  }

  receive_string(ca_readcap, IBPCA_CAP_LEN, fd);
//  strcpy(ca_readcap, lc_buffer);

  close(fd);
```

```c
  return(0);
}

/*-----------------IBP_CA_allocate-------------------*
  Description:
  IBP_CA_allocate allocates maxsize bytes of storage into
  the depot. The attributes of the storage are defined by
  attributes.  The duration must be finite, and the data
  type must be byte-array.

  Returns:
  0 on succesful completion, -1 otherwise
 *---------------------------------------------------*/
int             IBP_CA_allocate(    IBP_depot depot,
                                    IBP_timer timeout,
                                    ulong_t maxsize,
                                    IBP_attributes attributes,
                                    IBP_CA_cap writecap)
{
  int li_return, fd, ok;
  time_t lt_now, lt_lifetime;
  char lc_buffer[CMD_BUF_SIZE];
  IBP_CA_packet packet;

  /*
   * check IBP attributes
   */
  time(&lt_now);
  if ( attributes == NULL ) {
    fprintf(stderr, "error: invalid attributes\n");
    return(-1);
  }
  if ( attributes->duration > 0) {
    lt_lifetime = attributes->duration - lt_now;
    if ( lt_lifetime < 0){
      fprintf(stderr, "error: can\'t have negative lifespan\n");
      return(-1);
    }
  } else {
    fprintf(stderr, "duration must be finite\n");
    return(-1);
  }

  if ((li_return = d_CheckAttrRely(attributes->reliability)) != IBP_OK) {
    fprintf(stderr, "error: invalid reliability\n");
    return(-1);
  }

  if( attributes->type != IBP_BYTEARRAY) {
    fprintf(stderr, "Data type must be bytearray\n");
    return(-1);
  }

  /*
   * connect to IBP server
   */
//  fd = request_connection(depot->host, depot->port);

  fd = ca_connect_socket(depot->host, depot->port, timeout->ClientTimeout);
  if(fd == -1) {
    fprintf(stderr, "Connection to %s:%d failed\n", depot->host, depot->port);
    return(-1);
  }
//  send_ok(fd);

  send_packet(fd, IBP_CA_ALLOCATE, timeout->ServerSync, timeout->ClientTimeout,
              attributes->reliability, attributes->duration, maxsize, lt_lifetime,
              0, 0, NULL, NULL);

  ok = receive_int(fd);
  if(ok != IBP_OK) {
    fprintf(stderr, "IBP_allocate: %d\n", ok);
    close(fd);
//    return(-1);
    exit(1);
  }

  receive_string(writecap, IBPCA_CAP_LEN, fd);

  close(fd);
  return(0);
}

/*-----------------IBP_CA_store----------------------*
  Description:
  IBP_CA_store appends to a writecap previously obtained
  from IBP_CA_allocate.  The first size bytes of data are
  appended to the byte-array referenced by writecap.
  Readcap is then constructed from the entire byte-array's
  md5-checksum.

  Returns:
  0 on successful completion, -1 otherwise
 *---------------------------------------------------*/
int             IBP_CA_store(       IBP_CA_cap ca_writecap,
                                    IBP_timer timeout,
                                    char *data,
                                    ulong_t size,
                                    IBP_CA_cap readcap)
{
  char buffer[CMD_BUF_SIZE];
  int fd, ok;
  char writecap[IBP_CAP_LEN];
  MD5_CTX context;

  fd = create_ca_IURL(ca_writecap, timeout);

  if(fd == NULL)  {
    fprintf(stderr, "MD5 IBP Internal Error\n");
    return(-1);
  } else if(fd == -1) {
    return(-1);
  }

  send_packet(fd, IBP_CA_STORE, timeout->ServerSync, timeout->ClientTimeout,
              0, 0, size, 0, 0, 0, NULL, ca_writecap);

  ok = receive_int(fd);
  if(ok == -1) {
    fprintf(stderr, "Bad writecap\n");
    close(fd);
    return(-1);
  } else if(ok == -2) {
    fprintf(stderr, "Expired duration\n");
    close(fd);
    return(-1);
  }

//  writecap = (IBP_cap)malloc(IBPCA_CAP_LEN);
  receive_string(writecap, IBP_CAP_LEN, fd);
  context = receive_context(fd);
// printf("writecap is %s\n", writecap);

  ok = IBP_store(writecap, &timeout, data, size);
  if(ok == 0) {
    fprintf(stderr, "IBP_store error: %d\n", IBP_errno);
    send_int(fd, 0);
```

```
    close(fd);
    return(-1);
  }
  send_int(fd, ok);
  MD5Update(&context, data, size);
  send_context(context, fd);

  receive_string(buffer, IBPCA_CAP_LEN, fd);
  strcpy(readcap, buffer);
  close(fd);
  return(0);
}

/*------------------IBP_CA_load----------------------*
  Description:
  IBP_CA_load loads size bytes, starting at the offset
  position, from the byte-array accessed through readcap
  and stores the bytes into buf.

  Returns:
  Number of bytes read on successful completion, -1
  otherwise
 *--------------------------------------------------*/
int             IBP_CA_load(        IBP_CA_cap ca_readcap,
                                    IBP_timer timeout,
                                    char *buf,
                                    ulong_t size,
                                    ulong_t offset)
{
  int fd, ok, length;
  IBP_cap readcap;

  fd = create_ca_IURL(ca_readcap, timeout);

  if(fd == NULL)  {
    fprintf(stderr, "MD5 IBP Internal Error\n");
    return(-1);
  } else if(fd == -1) {
    return(-1);
  }

  send_packet(fd, IBP_CA_LOAD, timeout->ServerSync,
              timeout->ClientTimeout, 0, 0, size, 0, offset,
              0, NULL, ca_readcap);

  ok = receive_int(fd);
  if(!ok) {
    fprintf(stderr, "Expired duration\n");
    return(-1);
  }

  length = receive_int(fd);
  readcap = (IBP_cap)malloc(IBP_CAP_LEN);
  receive_string(readcap, IBP_CAP_LEN, fd);

  close(fd);
  if(size + offset > length) {          // ?? !! size - offset ??
    fprintf(stderr, "Invalid size/offset\n");
    return(-1);
  }
  length = 0;
  length = IBP_load(readcap, &timeout, buf, size, offset);
  if(length == 0) {
    fprintf(stderr, "IBP_load error: %d\n", IBP_errno);
    return(-1);
  }

  return length;
}

/*--------------------------------------------------*
  Description:
  Allows the user to perform the following operations on
  a storage area:
  1. probe a write or read cap,
  2. extend time limit
  3. extend size
  4. delete a writecap

  operations 2,3, and 4 require a writecap while operation
  1 can be used with a readcap or a writecap

  Returns:
  0 on successful completion, -1 otherwise
 *--------------------------------------------------*/
int  IBP_CA_manage( IBP_CA_cap man_cap,
                    IBP_timer timeout,
                    int cmd,
                    ibp_probe_info *status)
{
  int fd, ok;
  char lc_buffer[CMD_BUF_SIZE];

  if(strstr(man_cap, "/W/") != NULL)
    fd = create_ca_IURL(man_cap, timeout);
  else
    fd = create_ca_IURL(man_cap, timeout);

  if(fd == NULL)  {
    fprintf(stderr, "MD5 IBP Internal Error\n");
    return(-1);
  } else if(fd == -1) {
    return(-1);
  }

  send_packet(fd, IBP_CA_MANAGE, timeout->ServerSync,
              timeout->ClientTimeout, status->attrib.reliability,
              status->attrib.duration, status->maxSize, 0, 0, cmd,
              NULL, man_cap);

  if(cmd == IBP_CA_MANAGE_PROBE) {
    status->maxSize = receive_int(fd);

    if(status->maxSize == -1) {
      status->maxSize = status->exists = 0;
      fprintf(stderr, "Bad read/writecap\n");
      close(fd);
      return(-1);
    } else if(status->maxSize == -3) {
      status->maxSize = status->exists = 0;
      fprintf(stderr, "Expired duration\n");
      close(fd);
      return(-1);
    } else {
      status->exists = 1;
      status->attrib.duration = receive_int(fd);
    }

  } else {
    ok = receive_int(fd);

    if(ok == -1) {
      fprintf(stderr, "Bad writecap\n");
      close(fd);
      return(-1);
    } else if(ok == -2) {
```

```
      fprintf(stderr, "Must have a write-cap to change storage\n");
      close(fd);
      return(-1);
    } else if(ok == -3) {
      fprintf(stderr, "Expired duration\n");
      close(fd);
      return(-1);
    }
    ok = receive_int(fd);

    if(ok != IBP_OK){
      fprintf(stderr, "IBP_manage: %d\n", ok);
      close(fd);
      return(-1);
    }
  }

  close(fd);
  return(0);
}

/*-----------------------create_ca_IURL---------------------*
  Inputs: a capability
  Returns: a file descriptor

  Description: Extracts the hostname and port from the input
    capability, connects to the IBP_CA server at that
    location, and returns a file descriptor of the connection
 *---------------------------------------------------------*/
int create_ca_IURL(char *cap, IBP_timer timeout)
{
  int  fd, port;
  char *tmp, *chr, cap_type;
  char *host, type[6];

  if( cap == NULL || strlen(cap) <= 0) {
    fprintf(stderr, "Invalid Cap\n");
    return(NULL);
  }

  tmp = strdup(cap);
  ltrim_string(tmp);

  host = strstr(tmp, "//");
  host += 2;
  chr = strchr(host, ':');
  chr[0] = '\0';
  tmp = chr + 1;

  port = atoi(chr + 1);

  chr = strchr(tmp, '/');

  tmp = chr + 1;
  cap_type = tmp[0];

  if(cap_type == 'R') {
    strcpy(type, "READ");
  } else if(cap_type == 'W') {
    strcpy(type, "WRITE");
  } else {
    fprintf(stderr, "must be a read or write cap\n");
    free(tmp);
    return(NULL);
  }
  fd = 0;

//   fd = request_connection(host, port);
  fd = ca_connect_socket(host, port, timeout->ClientTimeout);
  if(fd == -1) {
    fprintf(stderr, "Connection to %s:%d failed\n", host, port);
    return(-1);
  }

//  send_ok(fd);

  return(fd);
}

/*-------------------------send_packet-------------------------*
 Inputs: a file descriptor, IBP_CA command, Timeout information,
         data attributes, size, offset, IBP_CA_manage command, data
         checksum, and a capability

 Description:
 The inputs to this function cover any IBP_CA command.  Everything
 is sent to the IBP_CA server.If the specified command does not
 require some of the inputs, then they are ignored.
 *-------------------------------------------------------------*/
void send_packet(int fd, int Cmd, int ServerSync, int ClientTimeout,
                 int reliability, int duration, int size, int lifetime,
                 int offset, int man_cmd, char *checksum, char *cap)
{
  IBP_CA_packet packet;

  packet.Cmd = htonl(Cmd);
  packet.ServerSync = htonl(ServerSync);
  packet.ClientTimeout = htonl(ClientTimeout);
  packet.reliability = htonl(reliability);
  packet.duration = htonl(duration);
  packet.size = htonl(size);
  packet.lifetime = htonl(lifetime);
  packet.offset = htonl(offset);
  packet.man_cmd = htonl(man_cmd);
  if(checksum != NULL) strncpy(packet.checksum, checksum, CHECKSUM_SIZE);
  if(cap != NULL) strncpy(packet.cap, cap, IBPCA_CAP_LEN);

  if( write(fd, (void *)&packet, sizeof(packet)) != sizeof(packet)) {
    perror("write()");
    exit(1);
  }
}

int ca_connect_socket(char *host, int port, int ClientTimeout)
{
  int sock, nfds, ok, err, val, len, starttime, success;
  fd_set writeset;
  struct timeval *timeout;
  struct sockaddr_in sn;
  struct hostent *he;

  starttime = time(0);

  success = 0;
  while(!success) {

    if(ClientTimeout == 0)
      timeout = NULL;
    else {
      timeout = (struct timeval *)malloc(sizeof(struct timeval));
      starttime = time(0);
      timeout->tv_sec = ClientTimeout - (time(0) - starttime);
      timeout->tv_usec = 0;
    }
```

```
    if(!(he = gethostbyname(host))) {
      fprintf(stderr, "%s: can't gethostname\n", host);
      exit(1);
    }

    sn.sin_family = AF_INET;
    sn.sin_port = htons((short)port);
    sn.sin_addr.s_addr = *(u_long *)(he->h_addr_list[0]);

    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
      perror("socket()");
      exit(1);
    }

    ok = connect(sock, (struct sockaddr *)&sn, sizeof(sn));

    if(ok < 0) {
      err = _get_errno();
      if (err != EINPROGRESS && err != EWOULDBLOCK) {  // and != SYS_OK  (defined as E
RROR_SUCCESS) ?? !!
        fprintf(stderr, "SocketConnect (), connect() failed\n\terror: %d, %s\n", errno
, strerror(errno));
        return(-1);
      }

      nfds = sock + 1;

      FD_ZERO(&writeset);
      FD_SET(sock, &writeset);

      if(ok = select(nfds, NULL, &writeset, NULL, timeout) == 0) {
        close(sock);
        fprintf(stderr, "ClientTimeout expired before a succesful connection was estab
lished\n");
        return(-1);
      } else if( ok < 0) {
        close(sock);
        perror("select");
        return(-1);
      }

      len = sizeof(val);
      if( getsockopt(sock,SOL_SOCKET,SO_ERROR,&val,&len) != 0 ){
        perror("getsockopt");
        return(-1);
      }

      if(val != 0){
        close(sock);
        if ( timeout > 0 && (starttime + ClientTimeout >= time(0)) ) {
          fprintf(stderr, "ClientTimeout expired before a succesful connection was est
ablished\n");
          return(-1);
        }
        success = 0;
      } else
        success = 1;
    } else
      success = 1;
  }
  return(sock);
}
```

```
#include "ibp_ComMod.h"
#include "ibp-lib.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>
#include "ibp_net.h"
#include "jrb.h"
#include "ibp_errno.h"

#include "ibp_base.h"
#include "ibp_protocol.h"
#include <md5.h>

#define IBP_CA_STORE              13
#define IBP_CA_ALLOCATE           14
#define IBP_CA_STORE_BLOCK        15
#define IBP_CA_LOAD               16
#define IBP_CA_MANAGE             17

#define IBP_CA_MANAGE_PROBE       18
#define IBP_CA_MANAGE_TIME        19
#define IBP_CA_MANAGE_SIZE        20
#define IBP_CA_MANAGE_DEL         21

#define CHECKSUM_SIZE             33

#define IBPCA_OK                   1

#define BAD_READ_WRITE_CAP        -1
#define NEED_WRITE_CAP            -2
#define DURATION_EXPIRED          -3

#define IBP_CAP_LEN CMD_BUF_SIZE
#define IBPCA_CAP_LEN 305

typedef char *IBP_CA_cap;

typedef struct {
  char host[256];
  char ca_host[256];
  int  port;
  int  ca_port;
} ibp_ca_depot;

typedef struct {
  int size;
  int maxSize;
  int exists;
  struct ibp_attributes attrib;
} ibp_probe_info;

typedef struct {
  int Cmd;
  int ServerSync;
  int ClientTimeout;
  int reliability;
  int duration;
  int size;
  int lifetime;
  int offset;
  int man_cmd;
  char checksum[CHECKSUM_SIZE];
  char cap[IBPCA_CAP_LEN];
} IBP_CA_packet;

socklen_t ibp_ca_addrlen;
ibp_ca_depot ca_depot;

int  IBP_CA_store_block(  IBP_depot,
                          IBP_timer,
                          ulong_t,
                          char *,
                          IBP_attributes,
                          IBP_CA_cap);

int  IBP_CA_allocate(     IBP_depot,
                          IBP_timer,
                          ulong_t,
                          IBP_attributes,
                          IBP_CA_cap);

int  IBP_CA_store(        IBP_CA_cap,
                          IBP_timer,
                          char *,
                          ulong_t,
                          IBP_CA_cap );

int  IBP_CA_load(         IBP_CA_cap,
                          IBP_timer,
                          char *,
                          ulong_t,
                          ulong_t);

int  IBP_CA_manage(       IBP_CA_cap,
                          IBP_timer,
                          int,                  // manage command
                          ibp_probe_info *);
```

```
#include <stdio.h>
#include <stdlib.h>
#include "ibpca_io.h"

/*----------------------------send_string--------------------------*
  Description: sends a string to a file descriptor

*-----------------------------------------------------------------*/
void send_string(char *s, int fd)
{
  int len, ok;

  len = strlen(s);
  send_int(fd, len);

  ok = write(fd, s, len);
  if(ok < 0) {
    perror("write");
    pthread_exit(0);
  }

}

/*----------------------------receive_string------------------------*
  Description: receives a string from a file
descriptor

*-----------------------------------------------------------------*/
void receive_string(char *s, int size, int fd)
{
  int len, ok;

  len = receive_int(fd);

  if (len > size-1) {
    fprintf(stderr, "Receive string: string too small (%d vs %d)\n",
                    len, size);
    exit(1);
  }
  ok = read(fd, (void *)s, sizeof(char) * len);
  if(ok < 0) {
   printf("READ read returned %d\n", ok);
   perror("read");
   pthread_exit(0);
  }
  s[len] = '\0';
}

/*----------------------------send_int-----------------------------*
  Description: sends an integer to a file descriptor

*-----------------------------------------------------------------*/
void send_int(int fd, int i)
{
  int h;

  h = htonl(i);
  write(fd, (void *)&h, sizeof(h));

}

/*----------------------------receive_int--------------------------*
  Description: receives an integer from a file
descriptor

*-----------------------------------------------------------------*/
int receive_int(int fd)
{
  int i;

  read(fd, (void *)&i, sizeof(int));

  return (ntohl(i));

}

/*----------------------------send_context-------------------------*
 Inputs: a md5 context and a file descriptor

 Description: sends a context to the file descriptor

*-----------------------------------------------------------------*/
void send_context(MD5_CTX context, int fd)
{
//  char buf[sizeof(MD5_CTX)];
  char *buf;

  buf = (char *)malloc(sizeof(MD5_CTX));
  buf = memcpy((void *)buf, (void *)&context, sizeof(MD5_CTX));
  write(fd, (void *)buf, sizeof(MD5_CTX));

  free(buf);
}

/*----------------------------receive_context----------------------*
 Inputs: a file descriptor
 Outputs: a md5 context

 Description: reads a context from the input file
descriptor
*-----------------------------------------------------------------*/
MD5_CTX receive_context(int fd)
{
//  char buf[sizeof(MD5_CTX)];
  char *buf;
  MD5_CTX *tmp, context;

  buf = (char *)malloc(sizeof(MD5_CTX));
  read(fd, (void *)buf, sizeof(MD5_CTX));

  tmp = (MD5_CTX *)malloc(sizeof(MD5_CTX));
  tmp = memcpy((void *)tmp, (void *)buf, sizeof(MD5_CTX));
//  memcpy((void *)&tmp, (void *)buf, sizeof(MD5_CTX));

  context = *tmp;

  free(buf);
  free(tmp);
  return context;
}
```

```
#include <md5.h>

void send_string(char *, int);
void receive_string(char *, int, int);
void send_int(int, int);
int receive_int(int);
void send_context(MD5_CTX, int);
MD5_CTX receive_context(int);
```