Content-Addressable IBP – Rationale, Design and Performance

Rebecca L. Collins James S. Plank

Department of Computer Science
University of Tennessee
Knoxville, TN 37996

[rcollins,plank]@cs.utk.edu

Technical Report UT-CS-03-512
Department of Computer Science
University of Tennessee
December 2003

Abstract

This paper describes an extension to the Internet Backplane Protocol (IBP), called Content-addressable IBP (IBPCA). IBP is an important protocol in distributed, Web, Grid and peer-to-peer computing settings, as it allows clients in these settings to access, manipulate and manage remote storage depots in a scalable and fault-tolerant fashion. Content-addressability adds the ability to reference storage by hashes of its contents. In this paper, we discuss the rationale behind IBPCA, important design decisions, and performance implications.

Keywords

Network storage, Grid storage, Peer-to-peer storage, content-addressability, Internet Backplane Protocol

1 Introduction

In distributed, Web, Grid, and peer-to-peer applications, the management of storage is of paramount importance. Storage location, ownership, capacity and reliability all impact the performance of these applications, and storage solutions based on extensions to legacy file systems [5, 8, 13] are too limiting for one or more of these dimensions. Logistical Networking [3, 11] has been proposed and developed as a way of integrating storage into networking according to classical end-to-end principles [12], so that all of the above properties are addressed in a cohesive and effective way.

At the base of Logistical Networking is the Internet Backplane Protocol (IBP) [11]. IBP is server daemon software and a client library that allows storage owners to insert their storage into the network, and allows clients to allocate and use this storage. The unit of storage is a time-limited, append-only byte-array. With IBP, byte-array allocation is like a network **malloc()** call — clients may request an allocation from a specific IBP storage server (or *depot*), and if successful, are returned trios of cryptographically secure text strings (called *capabilities*) for reading, writing and management. Capabilities may be used by *any* client in the network, and may be passed freely from client to client, much like a URL.

IBP does its job as a low-level storage service. It abstracts away many details of the underlying physical storage layers: block sizes, storage media, control software, etc. However, it also exposes many details of the underlying storage, such as network location, network transience and the ability to fail, so that these

may be abstracted more effectively by higher-level software. A suite of such software, collectively labeled the Logistical Runtime System (LoRS) has been built on top of IBP, providing very powerful functionalities that have been employed for peer-to-peer video storage and delivery [1], distributed software deployment [4] distributed generation of scientific visualization [7], and distributed text mining [10].

In this paper, we detail an extension to IBP, called *Content-addressable IBP (IBPCA)*, in which the handles to IBP byte-arrays contain hashes of the byte-arrays' contents. We discuss the rationale behind it, the important design decisions, and some performance benchmarks. The goal of the paper is to argue that a content-addressable storage substrate such as IBPCA can be useful as a low-level storage service, providing additional functionality (as compared to IBP) with performance that is only slightly degraded, and in some cases drastically improved.

2 IBPCA Rationale

The rationale behind the design of IBP has been well documented [3, 11]. Time-limited byte-arrays give storage owners a degree of autonomy and provide a clean failure model. The append-only nature of byte arrays eases synchronization problems, obviating the need for problematic distributed data structures such as mutexes and locks. The text capabilities allow clients to pass handles to data among themselves without registering with a central authority, thus easing scalability.

Adding content-addressability to IBP makes sense for several reasons. First, IBP does not ensure that the data returned to a client is correct. It is up to the client to perform this check in an end-to-end manner [3]. The LoRS tools perform these checks explicitly by allowing clients to store MD5 checksums with the metadata (called an *exNode*) that aggregates IBP byte arrays into distributed files. This is wasteful of space, since both the IBP capability and its checksum must be stored in the exNode. With IBPCA, the capability itself contains the checksum, allowing the client to perform correctness checks on the data without requiring extra metadata.

Continuing in this vein, a standard strategy for performing fault-tolerance in IBP is to replicate data among multiple depots [2]. This requires the exNode to store a different capability for each replica. With IBPCA, the exNode can store the checksum for a piece of data, the locations of the depots that hold each replica, and *no* capabilities. Again, this helps keep exNodes small.

Finally, and perhaps most importantly, IBPCA clients can make use of the fact that they know whether a depot contains a piece of data. Specifically, when a client attempts to write a piece of data that already exists in the depot, the write returns instantly. This should be a great performance improvement. Moreover, a client may check to see if a depot holds a piece of data by constructing the appropriate read capability and attempting to read it. This can help the client discover which depots from a certain collection actually hold desired data, and can help the client decide whether more copies of the data should be stored in the collection.

There are two potential downsides to IBPCA. First, managing checksums will cause overhead compared to standard IBP. However, since many uses of IBP manage checksums anyway, this may not actually be a big cost. Second, the ability to probe IBP servers for content may be a negative with respect to issues such as copyright violation and distribution of illegal material. However, as with IBP, clients may choose to encrypt data before storing. Therefore, judicious use of IBPCA can certainly relieve the burden of these problems on the storage owners. Moreover, they may aid in the discovery of clients who breach copyright violation.

3 Design Decisions

There are two main design decisions with IBPCA. First is the client API, which has some subtleties, especially concerning allocation and storage of data. Second is the software architecture. We describe each below:

3.1 IBPCA API

The IBPCA API is best presented in reference to the IBP API, which is summarized in Figure 1.[1] IBP clients allocate byte arrays with **ibp_allocate()**, which returns three text capabilities of the form:

ibp://*hostname***:***port*/*key*/*WRM*

The capabilities are for reading, writing, and management, and are used for the **ibp_load()**, **ibp_store()**, and **ibp_manage()** subroutines, respectively. The *key* is generated randomly by the IBP server, and is different for each of the three capabilities. All IBP operations may fail for various reasons (e.g. time-limit expiration, server failure) and all support time-outs to tolerate network failures. As stated in the Introduction, IBP has been carefully designed to provide a scalable, fault-tolerant storage substrate for distributed, peer-to-peer and grid computing systems, and has been very successful in several applications. More detail on IBP's design and philosophy may be found in papers by Beck [3] and Plank [11].

Subroutine	Function
ibp_allocate	Allocates a byte array.
ibp_load	Reads data from a byte array.
ibp_store	Appends data to a byte array.
ibp_manage	Refreshes time limits, extends size, etc.

Figure 1: IBP API, briefly summarized.

The IBPCA API looks very similar to the IBP API, but has some subtle differences in functionality. For each of the IBP calls in Figure 1, there is an analogous IBPCA call. Additionally, there is an **ibpca_store_block()** call. Whereas IBP employs three types of capabilities (read, write and manage), IBPCA employs only two types (read and write), which have the following format:

ibpca://*hostname***:***port*/**R**/*MD5-checksum*
ibpca://*hostname***:***port*/**W**/*key*

Capabilities in IBPCA are smaller than in IBP – 305 bytes as opposed to 1024 bytes.

[1]The complete IBP API is available from `http://loci.cs.utk.edu`. The complete IBPCA API is available in [6].

Ibpca_allocate() allocates space for a byte-array, and returns a write capability to the client. Clients may then append to this byte-array by calling **ibpca_store()** on this write capability. Each of these calls returns a new read capability containing the MD5 checksum of the entire byte-array's contents. This has to happen since the **ibpca_store()** call modifies the byte-array's contents, and therefore its MD5 checksum. Clients may then call **ibpca_load()** on the read capabilities to read their contents. Finally, **ibpca_store_block()** is a call that allocates and stores a block of data in one step. It simply returns the read capability of the stored block; no write capability is needed because the data's size is already known, and the data will not be modified in the future. The failure semantics of IBPCA are the same as IBP, including time-outs in all calls to tolerate network failures.

To reiterate the major differences between IBP and IBPCA, they are reflected in the different uses of the capabilities. IBP returns read, write, and management capabilities when a new byte-array is allocated. In IBPCA, 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, a read capability cannot be returned until after some data has been stored. In addition, the contents of the 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. This feature is possible because the byte-arrays are constrained to be append-only.

Instead of using management capabilities as in IBP, IBPCA clients use read or write capabilities to manage byte-arrays. Providing a single management capability is insufficient because the byte-array can have several different read capabilities, all associated with different sizes, and the size associated with a given read capability cannot be increased because the read capability would then require an updated MD5 checksum. To deal with these complications, IBPCA requires a write capability for extending the duration or size of an allocated byte-array, and allows either a read or a write capability to be used for probing information about a byte-array.

3.2 Software Architecture

As a prototype implementation for content-addressability, 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 decision to design the IBPCA server as a separate process was made mostly for ease of implementation, but the disjunction of the two servers also provides some flexibility to their interface since one content-addressable server can potentially service multiple IBP servers.

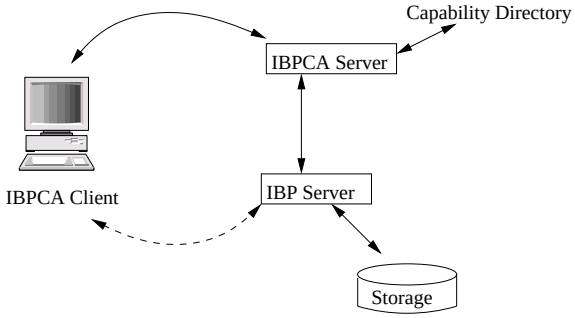

Figure 2: Basic IBPCA Architecture

The three communicating parties in the IBPCA architecture are the IBPCA server, the IBPCA client, and the IBP server. A diagram of the basic IBPCA architecture is shown in Figure 2. In general, the IBPCA client connects to the IBPCA server, which in turn connects the IBP server to process the IBPCA client's request. However, to reduce redundant data transfer over the network, the IBPCA client part of the API software connects directly to the IBP server whenever data must be transferred to or from an IBP depot. The IBPCA server regulates these data transfers since it maintains the directory of mappings of IBPCA to IBP capabilities.

To present a more detailed picture of the way the servers communicate, Figure 3 outlines the timeline of events in an **ibpca_store_block()** call. First, the IBPCA client calculates the MD5 checksum of the data, and then connects to the IBPCA server with an

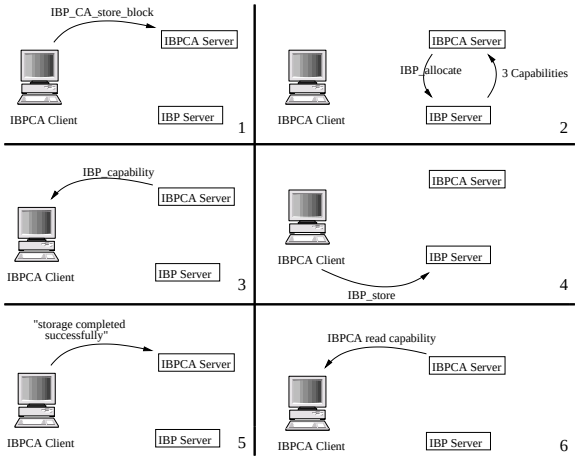

Figure 3: Communication path of **ibpca_store_block()**.

ibpca_store_block() call. If the IBPCA server determines that the data is already on the server, it notifies the client, which returns immediately with the read capability. Otherwise, the IBPCA server makes an **ibp_allocate()** call to the IBP server and receives the three IBP capabilities if the **ibp_allocate()** call is successful. Third, the IBPCA server sends the IBP write capability to the client so the client can store the data with an **ibp_store()** call. The client notifies the IBPCA server of the success of the **ibp_store()** call, and if it was successful, the IBPCA server updates its capability directories and sends the client an IBPCA read capability for future access of the byte-array.

Ibpca_store() and **ibpca_load()** both involve data transfer between the client and the IBP server. The interaction between the client and the servers in these two calls is very similar to the second half of the **ibpca_store_block()** communication path. First the client connects to the IBPCA server, and then the IBPCA server sends back the appropriate IBP capability so the client can connect to the IBP server to carry out the data transfer. **Ibpca_allocate()** and **ibpca_manage()** do not involve data transfer into an IBP depot, so all communication to the IBP server is handled through the IBPCA server.

Compared to IBP, IBPCA creates additional overhead in memory requirements, computation, and the latency of data transfer. Since the IBPCA and IBP

servers operate separately, the IBPCA server has no access to the internal IBP structures, and some information like size and duration of a byte-array must be recorded twice. In addition, the IBPCA server must store the IBP capabilities in order to map them to IBPCA capabilities. Since the IBP capabilities are relatively large, they account for the most of the memory overhead incurred by the IBPCA server. The computational overhead of IBPCA consists of calculating the MD5 checksums and processing client requests. The overhead of extra communication over the network between the IBPCA server and client typically outweighs the computational overhead.

4 Performance

To assess performance, we performed tests of the basic operations and their equivalents in IBP using one client located in Knoxville, TN, and two servers – a local one in Knoxville, and a remote one in San Diego, CA. For the local tests, the server 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 and 512-Mbytes RAM. Both machines were part of the Computer Science Department's local network (switched 100 megabit Ethernet) at the University of Tennessee. For the remote tests, the server ran on an AMD Athlon XP 2100+ 1733 MHz processor and 512-Mbytes RAM, located at the computer science department at the University of California, San Diego. The client ran on the same machine that was used for the local testing, and thus the communication between the two was over the commodity Internet.

All tests were performed on undedicated machines and networks. Therefore, all results presented are the average of over ten runs per data point. For each test, we employed three different byte-array sizes – 1 MB, 3 MB, and 20 MB. To present the data in a coherent way for these three sizes, the figures below (with the exception of Figure 4) display bandwidth in MB/sec rather than elapsed time.

Most of the IBPCA functions directly call their IBP counterparts, making their performance dependent on the performance of IBP. But since the overhead introduced by IBPCA is small, some of the test results show IBPCA slightly outperforming IBP. This discrepancy is most likely accounted for by variable outside network traffic during the testing, although it can also be related to the fact that communications between the IBPCA client and server are slightly different from communications between the IBP client and server, and communication between the IBP and IBPCA servers is fast since they are both on the same machine, while communication between clients and servers may be rather slow.

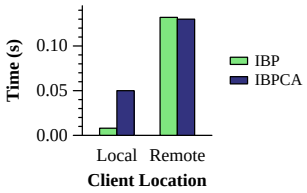

Figure 4: Performance of **ibp_allocate()** and **ibpca_allocate()** from local and remote clients.

Figure 4 shows that **ibp_allocate()** performs better over the local network, but the performance of the two functions over the remote network is very similar. Only one set of bars is displayed, because the performance of these allocation calls does not depend on the size of byte-array being allocated.

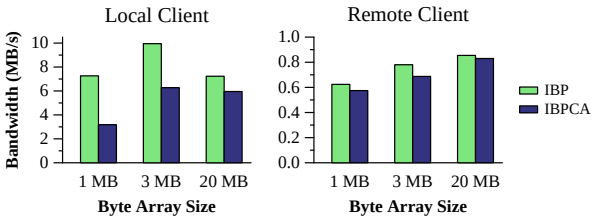

Figure 5: Performance of **ibp_store()** and **ibpca_store()** from local and remote clients.

Figure 5 compares the performance of **ibp_store()** and **ibpca_store()**. In each case, **ibpca_store()**'s performance is noticeably worse than **ibp_store()**'s. This is due to several factors, including the extra client/server interactions, and the fact that **ibpca_store()** must calculate the MD5 checksum of the data. Since these operations are roughly constant time, as compared to the transfer of the data from

client to server, IBPCA's performance relative to IBP is much better in the remote scenario (under 10% performance penalty), and improves as the size of the data increases.

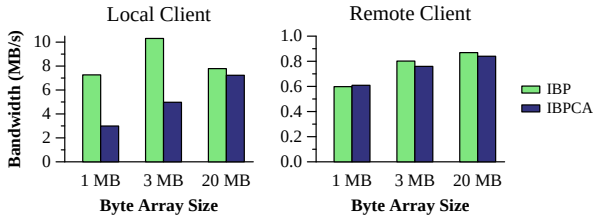

Figure 6: Performance of **ibp_store()**/**ibp_allocate()** and **ibpca_store_block()** from local and remote clients.

Figure 6 shows the performance of **ibp_store_block()**. Since no analogous IBP function exists, the performance is compared to the IBP calls of **ibp_allocate()** followed immediately by **ibp_store()**. Like **ibpca_store()**, the additional overhead is constant, so the difference in performance between the two functions decreases as the size of the data increases. Moreover, the performance of IBPCA compared to IBP is worse in the local area than in the remote scenario.

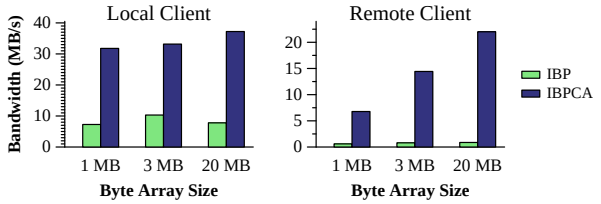

Figure 7: Performance of **ibpca_store_block()** with duplicate files from local and remote clients.

IBPCA has the feature that if a client makes an **ibpca_store_block()** call with data that is already stored at the server, the MD5 hash allows the server to recognize the duplicated data and return instantly with the read capability. Figure 7 measures this effect, showing the performance of **ibpca_store_block()** when a duplicate byte-array is sent to the server. Obviously, the benefits of detecting duplicate files increase dramatically as the size of the data increases, and are more dramatic in the wide area than in the local area.

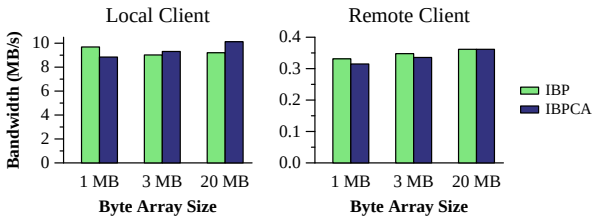

Figure 8: Performance of **ibp_load()** and **ibpca_load()** from local and remote clients.

Figure 8 shows the results of the **ibp_load()** and **ibpca_load()** tests. As would be expected, IBP and IBPCA perform almost identically for these tests.

5 Conclusion

In this paper we detail the architecture and performance implications of IBPCA. IBPCA extends IBP by embedding content-addressable hashes of data into the capabilities that are returned to the client.

Content addressable storage is desirable for a number of reasons. Content-addressable hashes provide a check that data returned to a client after storage is correct. Content-addressable capabilities are also smaller than traditional IBP capabilities and unless multiple appends are made, there are fewer capabilities overall to be dealt with. Not only do the content addressable capabilities allow the server to detect duplicate files, but they also allow the client to determine whether a file is already stored on a depot.

IBPCA offers a significant improvement in performance when the storage of duplicate files is requested through **ibpca_store_block()**. In all other cases IBPCA introduces a fixed overhead that decreases as the size of the data involved increases. This relationship suits IBP well since IBP's benefits are geared towards larger data. Moreover, since the LoRS tools already optionally incorporate MD5 hashes, the cost of built in content-addressability at the level of IBP can be offset by the removal of this option at the LoRS level.

We are currently extending this work in two directions. First, the IBP implementation team is embedding IBPCA functionality into the public release

of IBP. This will improve performance, since there will be no process boundary between the IBPCA and IBP servers. Second, we are collaborating with researchers from Intel Research Pittsburgh so that they can employ IBPCA as a page-storage service for their Internet Suspend/Resume project, which allows users to move their operating environment seamlessly from one location to another [9, 14].

6 Acknowledgements

This material is based upon work supported by the National Science Foundation under Grant Nos. EIA-0224441, ACI-0204007, ANI-0222945 ANI-9980203, and EIA-9972889 and the Department of Energy under grant DE-FC02-01ER25465. The authors thank Henri Casanova for providing access to the San Diego client, Micah Beck, Terry Moore, Scott Atchley and Stephen Soltesz for helpful discussions, and Michael Kosuch and Tom Bressoud for providing the initial impetus to initiate this project.

References

[1] S. Atchley, S. Soltesz, J. S. Plank, and M. Beck. Video IBPster. *Future Generation Computer Systems*, 19:861–870, 2003.

[2] S. Atchley, S. Soltesz, J. S. Plank, M. Beck, and T. Moore. Fault-tolerance in the network storage stack. In *IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems*, Ft. Lauderdale, FL, April 2002.

[3] M. Beck, T. Moore, and J. S. Plank. An end-to-end approach to globally scalable network storage. In *ACM SIGCOMM '02*, Pittsburgh, August 2002.

[4] B. Boehmann. Distributed storage in RIB. Technical Report ICL-UT-03-01, Innovative Computing Laboratory, March 2003.

[5] T. Clark. *Designing Storage Area Networks*. Addison-Wesley, Boston, 1999.

[6] R. Collins. Ibpca: Ibp with md5. Technical Report CS-03-511, Department of Computer Science, University of Tennessee, November 2003.

[7] J. Ding, J. Huang, M. Beck, S. Liu, T. Moore, and S. Soltesz. Remote visualization by browsing image based databases with logistical networking. In *SC2003 Conference*, Phoenix, AZ,, November 2003.

[8] H. Hulen, O. Graf, K. Fitzgerald, and R. W. Watson. Storage area networks and the High Performance Storage System. In *Tenth NASA Goddard Conference on Mass Storage Systems*, April 2002.

[9] M. Kosuch and M. Satyanarayanan. Internet suspend/resume. In *4th IEEE Workshop on Mobile Computing Systems and Applications*, Callicoon, NY, 2002.

[10] S. Y. Mironova and M. W. Berry *et al*. Advancements in text mining. In H. Kargupta, A. Joshi, K. Sivakumar, and Y. Yesha, editors, *Data Mining: Next Generation Challenges and Future Directions*. AAAI/MIT Press, Menlo Park, CA,, 2003.

[11] J. S. Plank, A. Bassi, M. Beck, T. Moore, D. M. Swany, and R. Wolski. Managing data storage in the network. *IEEE Internet Computing*, 5(5):50–58, September/October 2001.

[12] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems,*, 2(4):277–288, November 1984.

[13] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, and D. C. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39(4):447–459, April 1990.

[14] N. Tolia, T. Bressoud, M. Kozuch, and M. Satyanarayanan. Using content addressing to transfer virtual machine state. Technical Report IRP-TR-02-11, Intel Research Pittsburgh, August 2002.