DISP: Practical, Efficient, Secure and Fault Tolerant Data Storage for Distributed Systems

Daniel Ellard
Harvard University
ellard@eecs.harvard.edu

James Megquier
Gnuterra Corporation
jmegq@gnuterra.com

Abstract

We present DISP, a practical, efficient and secure client/server protocol for data storage and retrieval in a distributed environment and show how this protocol can tolerate Byzantine failure. We discuss variations on DISP that can be used as building blocks for different applications, and measure the performance of DISP on commodity hardware.

1 Introduction

This paper describes DISP, the *Distributed Information Storage Protocol*. DISP is a practical, efficient, secure and fault-tolerant client/server protocol for distributed data storage.

DISP is practical because it is simple to describe, easy to implement, and makes reasonable assumptions about the capabilities of the client and server. DISP is efficient in terms of network traffic; even in the presence of failures, the protocol requires transferring only marginally more than L bytes in order to retrieve an object of size L. DISP is secure because all of the data is stored and transferred in an encrypted form; compromise of the network or a server reveals none of the data. DISP is also able to ensure integrity of the data so that the data can be retrieved correctly even when servers have been corrupted.

DISP is novel in two ways: first, it requires no server-to-server or client-to-client communication. The only communication is between the clients and the servers. Protocols that require server-to-server communication in order to achieve consensus or agreement do not scale well; as the number of servers grows, the communication overhead and complexity of these protocols become daunting. In contrast, the upper bound on the number of messages required by each DISP operation is proportional to the number of servers, even in the worst case. The fact that DISP servers never communicate also means that it is easier to isolate server failures and prevent a Byzantine server from influencing correct servers.

The second novel aspect of DISP is its simplicity. DISP sacrifices some functionality and a small degree of performance and fault tolerance in favor of simplicity and ease of implementation. Garay et al. [8] and Alon et al. [1] describe systems that address many of the same issues as DISP and share some of its characteristics, but these systems use protocols and encoding schemes that are significantly more complex than DISP. Although these systems have desirable properties, we believe that their implementation would be substantially more difficult than that of DISP.

DISP manages the storage of immutable data objects by distributing the responsibility of storing each data object among a pool of autonomous and independently functioning servers. DISP does not implement the semantics of a file system, although it may be used as the storage manager for a distributed file system based on immutable storage, such as Venti [13]. Although DISP does not permit mutable objects, it does support versioning – there may be any number of versions of each object, and all versions are accessible.

Another important aspect of DISP is that it is parameterized; rather than being a single protocol tuned for one type of application, it is really a family of protocols that may be used for many applications.

DISP is scalable because nearly all of the computation required by the protocol is performed by the clients. We envision that clients will greatly outnumber servers, and therefore DISP is designed to minimize the amount of per-client overhead and computation required by the server.

DISP satisfies the BASE semantics described by Fox et al. [7]. Erasure-replication of data and tolerance of Byzantine server failures allow DISP clients to access the system as a whole even in the face of a large number of partial failures. Permission to write to servers is granted in a soft-state manner; if a client fails to complete the write protocol for any reason, it may simply initiate another write. Once a write has completed successfully, all clients will see a consistent view of the new version of the object.

In terms of the CAP principle [6], DISP sacrifices perfect consistency for high availability and resilience in the face of network partitions. Objects stored in a DISP system are accessible as long as a sufficient number of DISP servers that hold data related to the object are reachable and correct (and this number is a parameter chosen by the writer), and new objects may be added to the system as long as any subset containing this number of correct servers is reachable.

The rest of this paper is organized as follows: in Section 2 we describe the basic protocol and show how to extend this protocol so it can handle Byzantine failures in Section 3. Section 4 discusses the efficiency of DISP, and Section 5 gives example applications. In Section 6 we describe our implementation and benchmark its performance. In Section 7 we discuss related work and then conclude in Section 8.

2 An Overview of DISP

DISP is based on the INDIA protocol [5], which in turn is based on ideas from Rabin's description of possible applications of the Information Dispersal Algorithm (IDA) [14].

DISP is a client/server protocol. All of the data is stored on the servers. Unlike contemporary peer-to-peer (P2P) systems, which must be prepared to handle frequent changes in their constituency as peers enter and leave the system, we assume that the servers are available and stable most of the time. Therefore our protocol is optimized for the case where servers are available and behave correctly, although we do not require that this is always true. Clients, on the other hand, are not dedicated to the system, and may enter and leave the system at any point. The sets of client and server machines may overlap.

We assume that there exists a secure network infrastructure that allows participants in the protocol to perform mutual authentication and establish encrypted communication channels. We also assume that each server is able to sign messages in a manner that can be verified by any client or server.

Each data object is stored on a set of DISP servers called a *domain*. DISP supports any number of server domains, and each domain may support any number of clients.

In contrast to most P2P systems, which are designed to handle dynamic membership, DISP does not use a routing protocol to find servers. Instead, all hosts that are members of a DISP domain are given a DNS alias based on the name of the domain, and clients discover the list of servers in a domain via Secure DNS.

We begin with a description of IDA, which we use as the basis for our data encoding. We then introduce the central ideas of DISP by describing a simplified version of the protocol. We then present the extensions that complete the full protocol and show how it tolerates Byzantine failures.

2.1 The Information Dispersal Algorithm

The underlying representation scheme used by our system is Rabin's *Information Dispersal Algorithm* (IDA) [14]. IDA is an erasure code (also known as a forward error-correcting code) in which the original data is stored as a set of mutually redundant *shares* and then dispersed to different locations for safe storage.

The IDA coding scheme is analogous to the coding scheme used in RAID-5, where enough data to fill M disks can be encoded and stored on $M+1$ disks in such a way that the data can still be reconstructed if any single disk fails. The essential difference between IDA and typical RAID coding schemes is that RAID can usually tolerate the loss of only a single share of the data. IDA uses a more general scheme that permits the construction of an arbitrarily large number of distinct shares. We will use (M, N)-IDA to denote an IDA code for which N shares may be created for any data object, and any subset of M of these shares are sufficient to reconstruct the original data.

An (M, N)-IDA code has two properties that are relevant to our protocol:

1. If the original data has size L, then each share of an (M, N)-IDA encoding of the data has size L/M.

   Since each share has size L/M, and M shares are necessary to reconstruct, the total size of the shares needed to reconstruct an object of size L is $ML/M = L$.

2. An (M, N)-IDA code exists for any $M = p^n$ (where p is prime and $n \geq 0$) and N as long as $M \leq N$.

   If M is 1, then the IDA code degenerates into simply mirroring the data N times. $(N-1, N)$-IDA codes are isomorphic to parity or checksum-based codes, such as those typically used by RAID. The benefit of IDA comes when we desire N to be larger than $M+1$; it is possible to construct an IDA code with any desired degree of redundancy.

Both of these properties are proven in the original description of IDA [14].

Step 1: Obtaining a WriteHandle. The client sends a request to an arbitrary server in the domain, asking to create an object with a specific name, IDA code, and other attributes. If the request is valid, the server chooses a unique DISP object identifier (DOID) for the object, creates a *WriteHandle* for the DOID, and returns it to the user.

Step 2: Distributing the Shares. The client prepares shares of the object and sends the WriteHandle and one share to each server in the domain. Each server stores the share and acknowledges its receipt.

Step 3: Committing the Shares. Once the client has received acknowledgments from the distribution of its shares, it sends a request (which includes the WriteHandle) to each server to *commit* the share. Each server acknowledges when its share is committed.

Figure 1: The basic DISP write protocol.

2.2 The Write Protocol

The DISP write protocol consists of three steps, which are essentially a two-phase commit protocol with the client serving as the leader, as illustrated in Figure 1.

A WriteHandle is a certificate, signed by a server, that authorizes a specific client to store a share of the object, specified by the DOID, on each server in the domain. A WriteHandle includes the name of the object and any other object attributes specified by the client as well as the identity of the requesting client, the server, a per-server generation number, the IDA code, and the time.

Note that the IDA encoding is chosen by the writer, so the writer is free to choose whatever encoding they feel is most the appropriate for each object; all DISP servers must accept shares that use any code that is part of the DISP specification.

It is essential that the choice of server in the first step of this protocol is arbitrary. A WriteHandle signed by any server in the domain will be honored by any other server in that domain. Therefore, if any server that the client chooses is unavailable or incapable of granting a WriteHandle, the client is free to try another.

It is also important that the WriteHandle and DOID are transparent, so that the client can verify that the WriteHandle it receives in the first step corresponds to the object it requested. It is also essential that the WriteHandle contain the name of the client and be signed by the server. Because each party authenticates each communication, the servers will only accept shares from the client that was initially granted the WriteHandle. (A side-effect of this rule is that WriteHandles are usable only by a single client in the current specification of DISP; they may not be delegated or shared between clients.)

The client may attempt to subvert the protocol by requesting a WriteHandle with invalid attributes (such as ownership by a different client). A correct server will check the credentials of the client and only create a WriteHandle with the valid attributes. If the server and the client conspire to create a WriteHandle with invalid attributes, the resulting WriteHandle will not be accepted by any of the correct servers in the second step; no correct server accepts shares unless the WriteHandle has attributes that are valid for the client attempting the write. Each server can check the signature of the server who granted the WriteHandle to ensure that it has not been altered, and since clients must authenticate themselves to each server to which they write shares, each server can verify that the client attempting to use the WriteHandle is indeed the client to which the WriteHandle was granted.

In addition to resisting attacks from the clients, the basic write protocol is also resilient against several kinds of misbehavior among the servers. For example, the server contacted in the first step may refuse to grant a WriteHandle, attempt to delay the client by responding slowly, or return a WriteHandle for an object with different attributes than requested. The client can detect the latter situation by inspecting the WriteHandle to check that it matches the request. In all three situations, the client is free to repeat the request or try a different server. A correct server will respond quickly with a WriteHandle, and any WriteHandle created by any correct server will be accepted by all other correct servers. In the worst case, the client might not discover that the WriteHandle is invalid until its attempts to write shares to other servers are rejected.

A server cannot prevent a client from creating an object by creating an object with the same DOID (possibly with different attributes or different data). It may create an object with the same name, but because each DOID is marked with the identity of the writer (and correct servers will only accept shares of that object from that writer), it cannot create an object with the same DOID. Since no correct server will accept shares from a client other than the client named in the DOID and WriteHandle, the server cannot forge shares of the object and send them to other servers; the worst it can do is corrupt its own shares of objects.

2.3 The Read Protocol

The DISP read protocol consists of the two steps illustrated in Figure 2. Note that objects are created by name but shares may only be referenced via their DOID, and that it is possible for different versions of an object to share a name. When this happens, each server will return a list of all of the DOIDs that match the requested name and for which the client has authorization to read. It is the responsibility of the client to select the DOID of the version it wishes to access.

It may happen that a server does not respond quickly, or does not possess a share of the object that the client desires. In either case, the client may poll the other servers in the domain until it either finds enough shares to reconstruct, or discovers that no such object exists.

Note that DISP does not ensure that all observers of the system will see the same state, although the system will converge over time. Each client will observe a causally consistent view of the servers, but depending on the order in which the clients commit their shares or ask for ReadHandles, two concurrent clients may observe the system in different states. Clients are free to coordinate their activity so as to ensure that they observe the same contents of a DISP domain, but this protocol is not part of DISP at this time.

2.4 Other Operations

DISP also includes a delete operation, which removes an object from the system. (Note that the DOIDs are chosen in such a way that a new object will never share the DOID of any object, including deleted objects.)

Step 1: Requesting ReadHandles. The client sends a request to a server in a domain, asking for the *ReadHandles* of objects of the given name for which the server has a share. A server will only return ReadHandles corresponding to objects for which it has a complete and committed share.

Step 2: Gathering the Shares. The client sends the ReadHandle to M of the servers, and each server replies with their share of the corresponding object. As soon as M shares have arrived, the client reconstructs the object.

Figure 2: The basic DISP read protocol.

DISP includes two mechanisms that allow new servers to enter a domain and acquire shares of existing objects; which is used is a property of each object. The first method has the original writer take responsibility for creating the new shares, but in the second the ability to create new shares of an object is delegated to a trusted third party. The first method has the advantage that the client does not need to delegate its authority, but has the disadvantages that it requires the client to check periodically for new servers and that the client must perform a potentially huge amount of work. Further discussion of these protocols and their tradeoffs is part of our future work.

3 Tolerating Byzantine Failure

With the basic write protocol, the client will be able to write an object using an (M, N)-IDA code as long as at least M servers are available and correct. Similarly, the basic read protocol tolerates fail-stop failures as long as the number of participating servers does not fall below M. Unfortunately, the basic protocol does not tolerate Byzantine failure, even when more than M servers are correct. A Byzantine server may corrupt a share to prevent correct reconstruction, or may reveal the contents of the share to an unauthorized party. In this section, we show how to extend the basic protocol to address these issues.

We assume that the writer uses an (M, N)-IDA code and that there are $S \leq N$ servers in the domain. We can extend our protocols to handle F Byzantine failures, where $F < S/2$. Let Q be the size of a *quorum*. The client trusts any set of Q servers who supply consistent information about their shares. It follows that $Q \geq M$ in order to ensure that a reader can find enough shares to reconstruct, and $Q > F$ because otherwise F Byzantine servers could form a quorum.

The number of failures tolerated is at a maximum when $Q = \lceil (S+1)/2 \rceil$. However, the choice of Q is made on a per-operation basis. Depending on the priorities of the client, it might choose to disable fault tolerance altogether, tolerate a small number of failures, or pay the full cost of tolerating as many as $\lfloor S/2 \rfloor$ failures.

3.1 Ensuring Share Integrity

In the simple protocol, a client reading an object must trust that the shares returned by each server are identical to the shares that were written, and that those shares are mutually consistent. If one server returns a share that has been modified in any way, the client has no way to detect this modification. To compound this problem, IDA shares are malleable and therefore a Byzantine server can alter the client's view of the data to a predictable value.

A common approach to the problem of data integrity is the use of signatures: the writer signs each share it creates, and each reader verifies the signature before accepting the share. This method is well-suited for applications with a small number of readers and writers, but does not scale gracefully, because of the problems associated with key management. For example, if the writer's signing key is compromised, there is no easy way to revoke the key, issue a new key, and update all of the relevant shares. If the key is lost, there is no way to check the signature at all.

Another approach is to store a message digest of the original object with each share, so that the reader can check whether the reconstruction matches the original. Unfortunately, this method is only able to detect when reconstruction has failed and is not helpful in finding corrupted shares – the only way to find the correct subset of shares is by trial and error.

To avoid both of these problems, we use an approach based on the idea of *check vectors* as proposed by Krawczyk [11]. We use a cryptographic hash function to compute a check vector consisting of the message digests for every share of the object and then store the check vector with every share, and modify the reconstruction step as shown in Figure 3.

Once the reader has M shares, reconstruction proceeds as normal. As long as there are $Q \geq M$ and $Q > F$, the reader will be able to find the correct value of the check vectors and thence find a set of M consistent shares and reconstruct. (The most that the Byzantine servers can accomplish is to produce a group of $Q - 1$ consistent check vectors.)

One drawback with this approach is that a mischievous server can cause the reader to transfer a corrupt share in its entirety before the reader can determine that the share is corrupt and discard it in the third step. The root of this problem is that a single bit error (in either the check vector or the share) will invalidate the entire share. To bound the amount of information that can be invalidated by a single error, we use a block-oriented variation of IDA and check vectors. This makes it possible to isolate errors and reduces the amount of new data requested in the third step to a single block per error. A Byzantine server can still cause the reader to do extra work by adding errors to many blocks of the share, but the client is also free to choose which servers it uses, and therefore can adapt to this situation by preferring to gather shares in the first step from servers that have not recently provided bad blocks.

3.2 Share Privacy via Keyless Encryption

The writer must also trust that the server does not share information about its stored objects with unauthorized parties. Even servers that behave correctly and are physically secure when they are part of the system may divulge their secrets if an attacker can get a copy of their backups or discarded storage disks [9].

The general solution to this problem is to ensure that the data is always encrypted before it reaches storage. Unfortunately, implementing this is non-trivial for a large-scale system, for reasons similar to the problems of signatures: the decryption key may be lost, or it may be revealed. In the first situation, the data is lost

Step 1: The reader gathers Q check vectors from Q servers.

Step 2: If the reader discovers that not all Q check vectors are identical, it gathers additional check vectors from other servers until it has Q that match.

Step 3: The reader gathers M shares from the servers that provided the matching check vectors and compares each to the corresponding element of the check vector. If it is a match, then the share is retained. Otherwise, the share is discarded and the client requests a share from another server. This process repeats until the client has M shares that match the check vector.

Figure 3: Reconstruction with check vectors, for an (M, N)-IDA code and a quorum of size Q.

forever. In the second case, the data may be revealed to an unauthorized party. The compromised key must be revoked and the data re-encrypted with a new key, which authorized readers must then rediscover.

One method for defeating this attack is replace IDA with Shamir's secret-sharing scheme [16]. In this context, Shamir's method for sharing secrets is essentially to pad the data with randomly selected values and then use a slight variation of IDA to encode the result. Each resulting share reveals no information whatsoever about the value of the original data. Unfortunately, the amount of padding required is prohibitive; each of the resulting shares is the same size as the original data, so the total size of the shares needed to reconstruct an object will be M times larger than the object. More recent secret-sharing schemes, such as those described by Garay et al. [8], reduce this overhead considerably, but still increase the storage and network requirements. At this time, we feel that it is better to trade CPU for disk and network bandwidth.

We employ a very different solution: we encrypt each share with a randomly selected key, and encode the keys with a (Q, N)-IDA code and store the key shares along with the shares of the data. The key are of sufficient length that even if $Q-1$ key shares are known to the adversary, the remaining key space is large enough to provide the equivalent of a large key. To reconstruct, the client obtains Q key shares and reconstructs the key, and then obtains M data shares and uses the key to decrypt them, and then reconstruct.

Our approach is similar to one proposed by Herlihy et al. [10]. In our protocol, however, the keys are stored and reconstructed using the same mechanism as the shares, and we use a long key rather than a threshold scheme in order to prevent partial discovery of the key from a few of the shares.

4 The Cost of DISP

In this section we analyze the cost of DISP in terms of computation, number of messages, and total size. For this analysis, we assume that the DISP domain has $S = 2M - 1$ servers (of which as many as $F = M - 1$ may be Byzantine) and we are using an (M, N)-IDA code with a quorum size of $Q = M$. If the length of the data object is L, the length of each IDA share will be L/M, so the total size of the shares is $(2M - 1)L/M < 2L$.

The encryption, decryption, and fingerprinting steps are all $O(L/M)$ and so the cost of these steps for all of the shares together is proportional to $2L$. The IDA calculation, however, takes time proportional to $O(L)$ for each of the $2M-1$ shares, and thus the total amount of time required to compute the value of all of the shares is proportional to $SL = (2M-1)L$, and the time necessary to perform reconstruction is ML.

We have considered using different erasure codes in DISP in order to reduce the share computation overhead. Tornado codes are particularly promising, because of their relatively low computational requirements, and they have proven suitable for at least one DISP-like application [3]. For small values of M, however, IDA still performs reasonably well and lends itself to efficient implementation.

The total size of the check vectors is proportional to $S^2 = (2M-1)^2$, because each check vector contains the fingerprint of every share. For very large domains the size of the check vectors is a problem, but when S is reasonably small (i.e. $S < 16$) and L is more than several kilobytes, the shares constitute most of the data transferred by the protocol, and therefore the expected quantity of data transferred is only marginally more than L for reading and $2L$ for writing. Even when there are failures, the quantity of data transferred for reading is near L because of the block-oriented check-vectors: when a bad block is discovered, the reader can immediately terminate the transfer from the corresponding server and begin a transfer (beginning at the same block) from another. If the reader keeps track of which servers have provided bad blocks, the worst case is that each of the M Byzantine servers provides one bad block and then is ignored for the rest of the read operation. In this case, the reader has to read $M-1$ extra blocks. If β is the size of each block, then this gives a total of $L+\beta(M-1)$. If L is large and β and M are reasonably small, this value is close to L. In the worst case, β is equal to the share size, and $L+\beta(M-1) = L+(L/M)(M-1) < 2L$.

In the expected case, the write protocol requires one message exchange for the first step and $2M-1$ messages for each of the other two, for a total of $4M-1$. Similarly, the read protocol requires one exchange in the first step and $2M$ in the second (M exchanges to gather the $M = Q$ check vectors, followed by M exchanges to gather the shares), for a total of $2M+1$. In the worst case (when the client must contact the maximum number of servers per round in their search for a sufficient number of correct servers), a write may require $5M-2$ exchanges and a read may require $4M-1$.

An obvious optimization is to permit several operations to be piggy-backed in the same exchange; for example, combining the request to write a share of an object with the request to commit a share from another. Our implementation optimistically requests the share at the same time it requests the check vector for the share. This means that it may waste time transferring corrupt shares, but in the absence of failures the total number of message exchanges necessary for reading is reduced to $M+1$.

5 Applications of DISP

DISP is a family of protocols and permits the writer to choose which IDA code and which security and fault-tolerance options they desire on an object-by-object basis. This permits applications or system integrators to tailor DISP for their needs. In this section we outline several possible applications for DISP, and describe how DISP might be tuned for each.

Widespread Data Availability with Integrity: DISP can be used as the infrastructure for a system such as LOCKSS [15], whose goal is to disperse information in such a manner that it remains available and correct, even in the face of coordinated attacks [15]. For this application, a (1,*)-IDA code with check vectors and a modest quorum size is appropriate. This would ensure that there are many complete copies of each object and that each reader gets a correct copy.

Fault-Tolerant SAN: DISP can be used as the basis of a fault-tolerant SAN constructed out of commodity hardware. In this case, a (2,N)-IDA code would permit access to the data as long as at least two servers are functional. If the SAN is on an isolated network and all servers are trusted, then encryption of shares and messages can be omitted, increasing the potential throughput of the system considerably.

Media or Archival Servers: A variation of the fault-tolerant SAN is a pool of media or archival servers. By distributing the shares among a large pool of servers, we can do load balancing (and provide a degree of fault tolerance) without consuming excessive storage. For example, imagine that we have four movies stored on four servers. If each server stores one movie, then the server with the most popular movie may be overwhelmed while the others are idle. If the movies are dispersed via a (4,N)-IDA code, however, the load on each server will be equal no matter which movie is the most popular.

6 Performance Results

Our current implementation of DISP is written entirely in C and uses a home-grown RPC toolkit. We use SHA-1 to compute our check vectors, Blowfish for share encryption, and the OpenSSL implementation of SSLv3 (using RSA keys and RC4) for secure, authenticated communication.

DISP places no limitation on the number of servers nor the number shares necessary to reconstruct, or on which IDA codes are available. In our prototype, however, we chose to limit the number of servers per domain to 16, and we have only implemented three IDA schemes: (1,256)-IDA, (2,256)-IDA, and (4,256)-IDA. We use a fixed block size of 16K for share storage and check vector calculation. With a blocksize of 16K, the per-block overhead of the check vectors and key shares is less than 5%.

6.1 The Testbed

Our test client has a Pentium-III Xeon CPU running at 1.8GHz and a Alteon AceNIC gigabit copper Ethernet NIC. The test servers are a variety of Pentium-III machines with CPUs running at speeds ranging from 1.0GHz to 1.8GHz. All of the servers have an Intel PRO/1000 copper gigabit Ethernet NIC. The computational demands on the servers are relatively modest; the bottleneck for the servers is how fast they can read and write to their disks and communicate via SSL.

All the clients and servers run FreeBSD 4.8p13. We use `gcc` version 3.3.2, which generates significantly faster code than the default `gcc` (2.95.4) provided with FreeBSD 4.8.

6.2 Benchmark Results

Table 1 shows the throughput of the read protocol for several combinations of the protocol options. Performance is given as the av-

Protocol Options			IDA Code		
Secure Communications	**Check Vectors**	**Share Encryption**	(1,*) MB/s	(2,256) MB/s	(4,256) MB/s
No	No	No	47.6	52.8	36.3
No	Yes	No	30.4	25.7	21.1
No	No	Yes	15.4	14.0	12.5
No	Yes	Yes	11.8	10.9	10.0
Yes	No	No	17.0	16.0	13.8
Yes	Yes	No	13.3	12.1	10.8
Yes	No	Yes	9.4	8.7	8.0
Yes	Yes	Yes	7.9	7.4	6.9

Table 1: The effect of protocol options on read performance for three different IDA codes. The (1,256) is equivalent to mirroring; each share is an exact copy of the original data. For the (2,256) code, any two shares suffice to reconstruct the data, and for the (4,256) code, four shares are sufficient. Throughput is given as the average speed in MB/s for ten 100MB transfers. (The standard deviation for each average is less than 0.2 MB/s.) The time recorded for each transfer includes the time needed to establish the connection between the client and the number of servers necessary for each protocol. The quorum size $Q = M$ for the check vectors and share encryption.

erage speed in MB/s for ten 100MB transfers. The time measured for each transfer includes the time necessary to establish the connection and perform mutual authentication between the client and the servers. It does not include the time necessary to read the data from disk; we warm the cache with the contents of the shares before starting each benchmark. If we did not warm the cache, experience has shown that throughput would be limited by the speed of the slowest disk among our servers.

The fact that the (2,256) code runs more quickly than the (1,*) code in the first row of the table is not a fluke, but is an artifact of the testbed systems. In these two tests, almost exactly the same amount of data is transferred, but due to the limitations of our hardware and details of TCP flow control, it is possible for the client to read data from two hosts slightly more quickly than from one. This quirk is visible only because the network is the bottleneck for this particular situation. Our implementation uses several threads to pipeline the process of gathering share blocks and performing reconstruction; while one set of share blocks are being read from the network, other share blocks are being reconstructed. For these two cases, share reconstruction is faster than the network. In all other cases, as the computation increases, throughput drops.

Although the CPU in our testbed client is not particularly fast by contemporary standards, it is network-limited on a gigabit network for some tests. Even our slowest benchmark, at 6.9 MB/s, is within a factor of two of saturating a 100Mb/s link.

Performance drops when check vectors are enabled, but drops even more precipitously when share encryption and point-to-point secure communications are enabled. The high cost of encryption and SSL communication has been noted by other researchers [2]. We see no solution except the availability of faster ciphers and message digests, or better implementations, perhaps in hardware, of the the corresponding algorithms. The important point is that the IDA share reconstruction is fast enough on contemporary processors that the overhead of using IDA is dwarfed by the apparently unavoidable overhead of secure communication.

The time needed for the IDA computations and the time required by the data transfer for the write protocol is proportional to the time required by the read protocol; if the writer prepares and writes N shares, then the time required by these steps is roughly $\alpha N/M$ where M is the number of shares needed to reconstruct and α is the time required by the read protocol. The total time required by the protocol, however, is influenced by the amount of time that the server requires to write the shares to disk.

7 Related Work

Many of the ideas presented in this paper are not new; there has been a great deal of work on efficient and fault-tolerant data storage. Garay et al. and Alon et al. describe systems that share many of the characteristics of DISP [1, 8]. Both of these systems use protocols and encoding schemes that are significantly more complex than DISP. Both schemes have desirable properties, but it remains to be seen whether these protocols can actually be implemented in a practical manner.

DISP addresses many of the same issues as contemporary P2P systems such as OceanStore [12] and CFS [4], but differs in several crucial ways. First, DISP is not P2P, nor global in scale. Unlike DHT-based schemes that allocate resources in a pseudo-random manner across the entire set of machines in the system, we expect that the DISP world will be divided into many federated domains. Second, in DISP the client coordinates the activities of servers acting on its behalf, and there is no server-to-server communication. In the OceanStore protocol, a client communicates with the system via a single server, which in turn communicates with other servers via a Byzantine agreement protocol.

Weatherspoon et al. [17] provide an interesting comparison of erasure-replication versus copy-replication in terms of cost and failure modes. Weatherspoon identifies strengths and weaknesses of both kinds of replication, and supports our argument that DISP should provide seamless support for both in order to support as many applications as possible.

8 Conclusion

DISP is a practical, flexible and easy-to-implement protocol with good performance and fault-tolerance characteristics. It is simple (and therefore easy to analyze, implement and optimize) yet provides strong guarantees about data availability and integrity. DISP's flexibility follows from the fact that the clients choose the IDA code used to represent each object and the quorum size necessary for reconstruction, so they can make trade-offs between the total storage necessary to represent an object, the speed with which the object can be reconstructed, and the number of server failures the object can survive.

Future Work

We plan to port our prototype to a standard secure-RPC framework (instead of our home-grown library), and then restate our protocol in terms of this framework so that it can be implemented in a portable manner. At that point we will release both the full specification and our reference implementation as an open-source project.

References

[1] Noga Alon, Haim Kaplan, Michael Krivelevich, Dahlia Malkhi, and Julien P. Stern. Scalable Secure Storage when Half the System Is Faulty. In *Automata, Languages and Programming, 27th International Colloquium, ICALP 2000*, pages 576–587, July 2000.

[2] George Apostolopoulos, Vinod G. J. Peris, and Debanjan Saha. Transport Layer Security: How Much Does it Really Cost? In *Proceedings IEEE INFOCOM '99, The Conference on Computer Communications, Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies*, pages 717–725, March 1999.

[3] John W. Byers, Michael Luby, and Michael Mitzenmacher. Accessing Multiple Mirror Sites in Parallel: Using Tornado Codes to Speed Up Downloads. In *Proceedings IEEE INFOCOM '99, The Conference on Computer Communications, Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies*, pages 275–283, New York, NY, USA, March 1999.

[4] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-Area Cooperative Storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP 2001)*, pages 202–215, Chateau Lake Louise, Banff, Alberta, Canada, October 2001.

[5] Daniel Ellard, James Megquier, Lori Park, and Nina Yuan. The INDIA Protocol - Project Report. Technical Report TR-25-97, Harvard University DEAS, 1997.

[6] Armando Fox and Eric A. Brewer. Harvest, Yield and Scalable Tolerant Systems. In *Proceedings of the Seventh Workshop on Hot Topics in Operating Systems*, pages 174–178, 1999.

[7] Armando Fox, Steven D. Gribble, Yatin Chawathe, Eric A. Brewer, and Paul Gauthier. Extensible Cluster-Based Scalable Network Services, October 1997.

[8] Juan A. Garay, Rosario Gennaro, Charanjit S. Jutla, and Tal Rabin. Secure Distributed Storage and Retrieval. *Theoretical Computer Science*, 243(1–2):363–389, 2000.

[9] Simson L. Garfinkel and Abhi Shelat. Remembrance of Data Passed: A Study of Disk Sanitization Practices. *IEEE Distributed Systems Online*, 4(2), 2003.

[10] Maurice Herlihy and J. D. Tygar. How to Make Replicated Data Secure. In *CRYPTO '87*, pages 379–391, August 1987.

[11] Hugo Krawczyk. Distributed Fingerprints and Secure Information Dispersal. In *Proceedings of the 12th ACM Symposium on Principles of Distributed Computing*, pages 207–218, Ithaca, New York, USA, August 1993.

[12] John Kubiatowicz, David Bindel, Yan Chen, Steven E. Czerwinski, Patrick R.

Eaton, Dennis Geels, Ramakrishna Gummadi, Sean C. Rhea, Hakim Weatherspoon, Westly Weimer, Christopher Wells, and Ben Y. Zhao. OceanStore: An Architecture for Global-scale Persistent Storage. In *ASPLOS-IX Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 190–201. ACM, November 2000.

[13] Sean Quinlan and Sean Dorward. Venti: a New Approach to Archival Storage. In *Proceedings of the FAST '02 Conference on File and Storage Technologies*, pages 89–101, Monterey, CA, January 2002.

[14] Michael O. Rabin. Efficient Dispersal of Information for Security, Load Balancing, and Fault Tolerance. *Journal of the ACM*, 36(2):335–348, 1989.

[15] David S. H. Rosenthal and Vicky Reich. Permanent Web Publishing. In *Proceedings of the Annual USENIX Technical Conference, FREENIX Track (FREENIX'00)*, pages 129–140, June 2000.

[16] Adi Shamir. How to Share a Secret. *Communications of the ACM*, 22(11), November 1979.

[17] Hakim Weatherspoon and John Kubiatowicz. Erasure Coding versus Replication: A Quantitative Comparison. In *Peer-to-Peer Systems, First International Workshop, IPTPS 2002*, pages 328–338, March 2002.