



The India protocol - Project report

Citation

Ellard, Daniel, James Megquier, Lori Park, and Nina Yuan. 1997. The India protocol - Project report. Harvard Computer Science Group Technical Report TR-25-97.

Link

<http://nrs.harvard.edu/urn-3:HUL.InstRepos:35059734>

Terms of use

This article was downloaded from Harvard University's DASH repository, and is made available under the terms and conditions applicable to Other Posted Material (LAA), as set forth at

<https://harvardwiki.atlassian.net/wiki/external/NGY5NDE4ZjgzNTc5NDQzMGIzZWZhMGFIOWI2M2EwYTg>

Accessibility

<https://accessibility.huit.harvard.edu/digital-accessibility-policy>

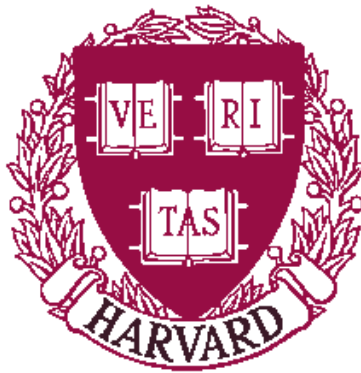
Share Your Story

The Harvard community has made this article openly available.
Please share how this access benefits you. [Submit a story](#)

The India Protocol - Project Report

Daniel Ellard
James Megquier
Lori Park
and
Nina Yuan

TR-25-97



Computer Science Group
Harvard University
Cambridge, Massachusetts

The India Protocol - Project Report

CS262 Project

Daniel Ellard, James Megquier, Lori Park, Nina Yuan

May, 1997

Contents

1	Introduction	1
1.1	A Word About IDA	1
1.2	Document Overview	2
2	India Protocols	3
2.1	The Read Protocols	3
2.1.1	Overview	3
2.1.2	The Simple Read Protocol	4
2.1.3	The Byzantine Read Protocol	5
2.2	The Write Protocol	6
2.2.1	Overview	6
2.2.2	The Simple Write Protocol	7
2.3	The Authentication Protocol	8
2.3.1	Overview	9
2.3.2	Assumptions	9
2.3.3	Security Requirements	10
2.3.4	General Protocol Description	10
2.3.5	Analysis	11
2.3.6	WhoAreYou Protocol	11
3	Enhancements to the India Protocols	13
3.1	Assumptions	13
3.1.1	Tools	13
3.2	The Write Proxy – Writing When Servers Are Down	13
3.3	Beating Sniffers – Secure Send and Deliver	14
3.3.1	Secure Send	14
3.3.2	Secure Deliver	14
3.3.3	Properties of Secure Send and Deliver	15
3.3.4	A Discussion of Public Keys	15

3.4	Quotas and Restrictions – Beating the Hoggy Client	15
3.5	Fingerprinting and Signatures – Beating the Hoggy Server	16
3.5.1	The Rude Server	17
3.6	Fingerprinting the Data and the Shares	17
3.6.1	The Fingerprinted Writing Algorithm	17
3.6.2	The Fingerprinted Reading Algorithm	19
3.7	Concealing the Share Database	20
3.8	Attacks With No Known Defense	21
3.8.1	Stolen Keys	21
3.8.2	The Devious Server	21
3.9	Conclusion	22
3.9.1	The Trusted Computing Base	22
4	Prototype Implementation	23
4.1	Basic types	24
4.1.1	Share	24
4.1.2	Handle	24
4.1.3	WriteHandle	24
4.2	The IDA	24
4.3	Server	25
4.4	Clients	25
5	Testbed Environment	27
5.1	Overview	27
5.2	Testbed Clients	27
5.2.1	iget	27
5.2.2	iput	27
5.2.3	ils	28
5.3	Testbed Servers	28
5.4	Elf Scripts	28
5.4.1	elf-config	29
5.4.2	elf-keyconf	29
5.4.3	elf-put	29
5.4.4	elf-rm	29
5.4.5	elf-svr-list	29
5.5	Test Scripts	30
5.5.1	test-speed	30
5.5.2	test-byz	30
5.5.3	troll-share	30

6	Benchmark Results	31
6.1	Throughput	31
6.1.1	Running Locally	31
6.1.2	Running Remotely - Slow Client	31
6.1.3	Running Remotely - Fast Client	32
6.1.4	Remarks	32
6.1.5	Compiled Java versus Interpreted Java	33
6.2	Correctness	33
7	Known Bugs	35
8	Future Work	37
A	Related Work	38

Chapter 1

Introduction

The goal of this project is to explore the potential of the IDA (Information Dispersal Algorithm) as the basis for an efficient, highly available, fault tolerant, and secure distributed data storage system. To this end, we have developed and implemented protocols that are significantly less complex than traditional replicated services, yet provide many of their benefits.

The distributed systems aspect of our project is the design of protocols with which clients and servers can maintain apparently consistent information without server-to-server communication.

The protocols we have developed for our system have the following features:

- *Simplicity*

The basic India protocols are easy to understand and implement. The enhanced protocols (see chapter 3 introduce some added complexity, but do not change the underlying nature of the protocol.

- *Scalability*

The computational complexity of the system (amount of computation, and the number of messages required for each operation) scales linearly with the number of clients and servers.

In addition, nearly all of the computation is performed by the clients, so each server can support a large number of clients. (We expect that servers will be bandwidth-limited rather than CPU-limited, and our benchmarks support this belief.)

Our prototype system, implemented in Java, demonstrates the feasibility of this approach.

1.1 A Word About IDA

IDA allows us to *disperse* a data object into many shares, and then *reconstruct* the object at a later time with a subset of the original shares. Since each server is only responsible for one share of each

object, the system can be robust in the case of server crashes, transient failures and corruption of data (since not all of the original shares are required for reconstruction) without the space overhead of replicating the original data one or more times. IDA also gives us fine-grained control in determining the optimal tradeoff between robustness and availability (by having more servers) and additional space overhead and time required to write objects (since more servers will need to be contacted to write more shares).

For more information on IDA itself, please refer to the original paper by Michael Rabin [7], or see <http://www.eecs.harvard.edu/~india/ida/>.

1.2 Document Overview

This section gives an overview of the organization of this document.

- Section 2 (page 3) describes the protocols used by our system for the reading and writing of objects and the mutual authentication of clients and servers.
- Section 3 (page 13) discusses vulnerabilities in our protocols as currently designed and implemented, and explores alternatives for removing or lessening these weaknesses.
- Section 4 (page 23) gives an overview of our implementation, describing the key classes and packages. For more information, the `javadoc` of our code is available via the India web page.
- Section 5 (page 27) describes the details of our testbed environment, including a description of the scripts that we use to configure and test our implementation.
- Section 6 (page 31) details the results of our performance benchmarks.
- Section 7 (page 35) lists the known bugs and shortcomings of our current implementation.
- Section 8 (page 37) discusses directions for future research.
- Appendix A (page 38) gives a survey of systems or research areas that are related to our system.

Chapter 2

India Protocols

The heart of the India system is the protocols that define how to securely read and write India objects (or more specifically, the shares into which objects are dispersed.) This section describes, in increasing order of complexity and sophistication, the two read protocols (Simple and Byzantine) and the write protocol (Simple) that we have defined and implemented in our prototype system. The authentication protocol is also described. This section includes brief notes on implementation details where appropriate, but for more information, please see Section 4.

2.1 The Read Protocols

2.1.1 Overview

The read protocol gathers the shares of an India object from the various servers and performs reconstruction. The read protocol framework consists of two main steps:

1. `getHandles`

In the first step, a Client asks a Server (or a set of Servers) for a set of Handles that correspond to a given ShortName, and the Server responds with such a list (possibly empty).

2. `getShares`

Once it has a handle for the desired object (possibly after choosing from a set of handles for objects with the same ShortName), the Client sends requests to several different Servers for their shares of the object that Handle represents.

Depending on the number of correct shares it receives, and the number required by the particular read protocol, the Client may then be able to reconstruct the original data object with the shares.

We have currently defined and implemented two read protocols: Simple and Byzantine¹ These protocols are described in greater detail below.

2.1.2 The Simple Read Protocol

The simple read protocol, as its name implies, is the foundation of the India read protocols. For an (M, N) IDA system, it asks for N shares, and must receive all N shares (correctly) in order to reconstruct the object.

To read an object from an India system with this protocol, the following exchange takes place:

1. **Client** sends a request to Server, asking for a list of all the Handles that the Server knows about that correspond to the given ShortName, by invoking the following function.

- *Function:* `Handle[] getHandles(ShortName)`

This is actually a client-side function that, in turn calls

`Handle[] getHandles(ShortName, servernum)`

on some server known to the client (the first server, in our implementation).

2. A **Server** receives the request for a list of Handles that correspond to the given ShortName.

- Does any object named ShortName in the ShareDb? If not, return an empty list.
- Otherwise, return a list of the objects named ShortName.

At this point in the protocol, Client has a list of Handles of objects with the given ShortName.

3. The **Client** chooses one of the Handles², and sends requests to a number of different share servers for the shares corresponding to that Handle, and gets enough shares to reconstruct. (Note that each server only has one share). For each server $Serv_i$:

- **Client** requests the value of share S_i , from $Serv_i$.
- **Server** $Serv_i$ delivers the request for the specified share and does the following (the execution of this step is atomic with respect to the local state of the server).
 - Is the handle well-formed? If **NO**, throw `IllegalArgumentException` and return.
 - Does an object exist with this ShortName? If **NO**, throw `IllegalArgumentException` and return.

¹A third class, `IndiaReadShamir`, extends `IndiaReadByz` to perform Shamir's shared secret computation scheme, but since this is an extension to `IndiaReadByz` and is not a separate protocol, it will be discussed in Section 3. A fourth class, `IndiaReadPause`, implements the `ReadSimple` protocol but is carefully instrumented to allow troll activity.

²In our current implementation, the client always chooses the first handle it finds; however, it could in principle determine which one it wants and use that.

- Has the object been marked as a stub? If **YES**, throw `StubShareException` and return.
- Get the betas. If there are any errors, throw `IndiaGeneralException`.

If the Client now has sufficient correct shares to reconstruct the object, it proceeds to do so.

There are several important failures that this protocol cannot prevent. Note that such failures are arbitrary, or Byzantine; simpler errors are handled (at least in some sense) by this protocol. These failures are discussed in chapter 3.

2.1.3 The Byzantine Read Protocol

The Byzantine read protocol adds some additional sophistication to the simple read protocol in order to deal intelligently with trolls or misbehaving servers. While the simple read protocol simply tries to get shares from N servers, the Byzantine read protocol tries to get shares from more than N servers, and then acts according to the number of shares received:

For instance, in our implementation of Byzantine Read for (F16, 2) IDA, the client requests $N + 2$ shares, with the following possible outcomes:

- If fewer than 2 shares arrive, then reconstruction is not possible.
- If exactly 2 shares arrive, then reconstruction is possible (as above), but error checking is not.
- If 3 shares arrive, then reconstruction with detection of errors but without correction of one troll is possible.
- If 4 or more shares arrive, then reconstruction with correction of one troll is possible.³

Because reconstruction may be possible with 2 or more shares, but our confidence in the validity of the reconstruction varies depending on the number of shares, in the actual implementation of the reading protocols the caller is given the option of finding out which of these possibilities occurred.

The protocol is described in detail below; note that it differs from the simple read protocol in the way it processes the received shares.

1. **Client** sends a request to Server, asking for a list of all the Handles that the Server knows about that correspond to the given ShortName, by invoking the following function.

- *Function:* `Handle[] getHandles(ShortName)`

2. **Server** receives the request for a list of Handles that correspond to the given ShortName.

- Does any object named ShortName exist?

³A more general Byzantine reconstruction algorithm is given in section 3.6.2.

- If **NO**, return an empty list.
 - If **YES**, return a list of all such objects.
3. **Client** chooses one of the Handles⁴, and sends requests to a number of different share servers for the shares corresponding to that Handle.
- **Client** requests the value of share S_i , from $Serv_i$.
 - **Server** $Serv_i$ delivers the request for the specified share.
 - Is the handle well-formed? If **NO**, throw `IllegalArgumentException` and return.
 - Does an object named `ShortName` exist? If **NO**, throw `IllegalArgumentException` and return.
 - Has the object been marked a stub? If **YES**, throw `StubShareException` and return.
 - Send the share to the client. If any errors occur at this point, throw `IndiaGeneralException`.
 - If fewer than N shares are received, reconstruction is not possible.
 - If N shares are received, reconstruction is possible but error checking is not.
 - If more than N shares are received, error detection and/or correction may be possible, depending on the specific IDA scheme used.

2.2 The Write Protocol

2.2.1 Overview

The Write Protocol is used to create new India data objects. At a high-level, the protocol consists of two exchanges between a client and the India servers:

1. `createHandle`

The client asks one server to create a new object, and receives a `WriteHandle` in return if successful.

2. `putShares`

The client disperses the shares of the `DataValue` among the India servers, using the `WriteHandle`.

Implementation note: The functions `createHandle` and `putShares` are actually composed of multiple Java methods, even though in this section they are described as integrated functions.

⁴Again, in our implementation, the first Handle it finds.

2.2.2 The Simple Write Protocol

1. **Client** sends a request to Server, asking for a WriteHandle for the given ShortName, by invoking the function:

- *Function:* WriteHandle createHandle(shortName)

2. **Server** receives request for a WriteHandle and performs the following operations until one is met:

- (a) Is the request authentic?

The first check should always be for authenticity; otherwise, an attacker could cause the server to waste CPU cycles performing computations or state changes that will need to be unrolled.

- (b) If **NO**, throw `IndiaAuthException` and return.

- (c) If **YES**, does the client have permission to create the object? (*Not currently implemented*).

- (d) If **NO**, throw `IndiaPermException` and return.

- (e) If **YES**, does an object with this name already exist?

Since our write protocol is currently immutable, this is the second most important check after testing for authenticity.

- (f) If **YES**, throw `AlreadyExistsException` and return.

- (g) If **NO**, can a stub object be created for this handle?

This is also part of the immutability check, since the failure of the creation of the stub object implies that a client, either this one or another, has already begun writing an object with the same ShortName, or that trolls have infested the server.

- (h) If **NO**, throw `IndiaGeneralException` and return.

- (i) If **YES**, create a Handle for this object. The Handle is a UUID that includes:

- ShortName, supplied by client.
- LongName, which is the ShortName concatenated with a server-generated “uniquifier”.
- CreationInfo, which is set to the server’s “key name”.

- (j) Sign the Handle, thus creating a WriteHandle for the object. The WriteHandle consists of the Handle and the server’s digital signature of the Handle.

- (k) Return the WriteHandle.

At this point, an empty stub object with the given name has been created on Server, and Client knows what the handle of that object is. None of the other servers know of the existence of this object; however, the Handle gives the client the ability to distribute shares of the object among all of the servers.

3. **Client** creates N shares, where N is the smaller of the field size and the number of reachable servers.
4. **Client** sends a request to each Server $Serv_i$ to store a share S_j of the object by invoking the function:
 - *Function:* `void putShare(WriteHandle, share)`
5. **Server** receives the request:
 - (a) Is the request authentic? If **NO**, throw `IndiaAuthException` and return.
 - (b) If **YES**, does the client have permission to create the object? (*Implementation Note:* This is not currently implemented).
 - (c) If **NO**, throw `IndiaPermException` and return.
 - (d) If **YES**, is the Handle valid (not null, and properly formed)?
 - (e) If **NO**, throw `IllegalArgumentException`.
 - (f) If **YES**, can the object be created?
 - (g) If **NO**, throw `IndiaGeneralException`.
 - (h) If **YES**, write the share.

There are several important failures that this protocol cannot prevent. These are discussed in chapter 3.

We have implemented three versions of this write protocol. `IndiaWriteSimple` is the ordinary version, while `IndiaReadShamir`, extends this protocol to use Shamir's shared secret computation scheme (see section 3.7). A fourth version, `IndiaWritePause`, implements the `WriteSimple` protocol but is carefully instrumented to allow troll activity.

2.3 The Authentication Protocol

Do not believe in anything simply because you have heard it. Do not believe in anything simply because it is spoken and rumored by many. Do not believe in anything simply because it is found written in your religious books. Do not believe in anything merely on the authority of your teachers and elders. Do not believe in traditions because they have been handed down for many generations. But after observation and analysis,

when you find that anything agrees with reason and is conducive to the good and benefit of one and all, then accept it and live up to it.

– The Buddha

2.3.1 Overview

The authentication protocol (also known as the WhoAreYou protocol) used in the India system provides for mutual authentication of the client and server to each other. It is not strictly a separate protocol, since it is integrated into the India read and write protocols, but is described more fully in this section than in the others. Differences between the protocol as described here and our implementation (due to the lack of certain cryptographic functions) are also mentioned briefly here.

2.3.2 Assumptions

The authentication protocol makes certain assumptions regarding the infrastructure available in the system.

1. We have clients and servers (collectively referred to as entities).

Implementation note: Ideally we would like to identify clients with process-level granularity, but this appears to be difficult in Java.

In our prototype implementation, we intended that the clients would be identified by their users, and this identity of the users would be specified on the command lines of the clients. This functionality is not fully implemented yet.

2. Each entity has a public/private key pair for digital signature purposes.

Implementation note: We use the Digital Signature Algorithm (DSA) classes in the `java.security` package to generate keypairs and to perform operations on them.

3. Each entity's key pair is verifiable via a public-key certificate infrastructure.

note: There are a variety of public-key infrastructure models that could be used, from hierarchical X.509 certificates to signed PGP keys in a web-of-trust system, to name but two. Creating even a simple true public-key infrastructure is outside the scope of this project, and thus for the purposes of our prototype, public keys are made available to all entities via files in known directories.

4. Each entity has a public/private key pair for key generation purposes.

Our protocol requires the ability for any pair of communicating entities to be able to generate a shared secret key. There are a variety of ways in which this can be done; the most popular methods are to use an asymmetric encryption algorithm such as RSA, or to use a key generation algorithm such as Diffie-Hellman.

Implementation note: Due to a lack of RSA and Diffie-Hellman packages for Java, this requirement is approximated in our prototype with a set of secret keys (random numbers) generated by an elf and placed in a set of known subdirectories.

2.3.3 Security Requirements

1. Each message⁵ exchanged between any pair of entities must satisfy the properties of data origin authentication and message integrity (i.e. we know who the message is from, and we know that the message was not altered in transit).
2. Thus, anything an entity must do on a per-message basis to achieve these properties must be fast and preferably add low space overhead to the message.

2.3.4 General Protocol Description

When entity A wants to send a message M to entity B , A does the following:

1. Check state; does it already have a shared secret with B ? (*Implementation Note:* Currently, the answer is always “yes.”)
 - If **NO**: Generate a shared secret: get and validate B ’s DH public key, and compute the secret key k_{AB} [8]. Add k_{AB} to state kept for B .
 - If **YES**: Compute $M_{auth} = \text{keyedHash}(M, k_{AB}, SN_{AB})$, where SN_{AB} is the sequence number for messages from A to B , and is incremented by A each time it sends a message to B . (M_{auth} is a known size depending on the hashing (fingerprinting) algorithm used, 20 bytes for SHA-1, 16 bytes for MD5.)
 - Send $M + SN_{AB} + M_{auth}$ to B (total overhead, 20-24 bytes, assuming a 32-bit sequence number).

When entity B receives a message M' from entity A , B does the following:

1. Decompose M' into $M + SN_{AB} + M_{auth}$ (since B knows the size of M_{auth} – the choice of hashing algorithm is fixed system-wide).
2. Check state; does it have a shared secret with $A(k_{BA})$?
3. If **NO**, compute as described above.

⁵Of course, the RMI API shields the programmer from the knowledge that actual messages are being sent over the wire to invoke the method of interest. For the purposes of this general authentication protocol section, however, it is more convenient to think of the actual packets being exchanged, as long as it is kept in mind that this will need to be translated into a more Java-esque form during design and implementation (as discussed later in this section).

4. If **YES**, compute $M'_{auth} = \text{keyedHash}(M, k_{BA}, SN_{AB})$.
5. If $M_{auth} = M'_{auth}$, then the message is OK. Otherwise, the message has been tampered with or forged.

2.3.5 Analysis

The requirement for a fast protocol implies the use of a keyed message authentication code (MAC) function⁶ or a symmetric encryption function instead of a digital signature function for ensuring message authentication and integrity. The use of the sequence number in the protocol ensures that a troll cannot replay previous requests.

The main cost of this protocol occurs at initialization time (or the first time an entity talks to a new other entity) since the computation of the secret key requires a (slow) public-key operation. Once that is completed though, each message requires only two (fast) cryptographic hashes (in computing the HMAC), and some state lookup on each end.

Each entity has to keep state (secret key and sequence number) for each party with which it communicates, so this is not terribly scalable⁷, particularly on the server end (assuming many clients, fewer servers). This is certainly not a problem for our prototype, but might be a problem if we want to address broader applicability of our system. It is not, however, any worse than a cookie-based system.

2.3.6 WhoAreYou Protocol

This section describes how this general authentication protocol fits into the read and write protocols in the India system. In the interest of expedient implementation, some aspects of the protocol have been simplified from the theoretical ideal; for instance:

- A simple envelope keyed hash (where the key is prepended and appended to the data to be hashed) is used instead of the HMAC (Hashed Message Authentication Code) that is generally favored in the cryptographic community today.
- The lack of either a freely-available Diffie-Hellman package or an RSA package for Java prevents us from truly generating a shared secret on the fly between two entities in an easy way. Thus, for the purposes of our prototype, `elf` (using `elf-keyconf`) generates a secret key (a random number) for each pairwise combination of entities in the system.

⁶HMAC is a currently accepted keyed MAC function that applies a cryptographic hashing function (like MD5 or SHA-1) twice to “ $M \text{ concat } K$ ” where M is the message to be protected, and K is a secret key shared only by the communicating pair.

⁷Of course, servers could cache a number of secrets they share with clients, and require re-authentication on a cache miss.

- A timestamp is used in place of a sequence number, for a (minimal) gain in ease of implementation.

The authentication protocol is invoked each time a client calls a server function (in our current implementation there are two such functions, `createHandle` and `putShare`.)

The `WhoAreYou` instantiation of our more general authentication protocol thus proceeds as follows:

1. Immediately preceding a call to `createHandle`, the client first calls the server's `whoAreYou` method: `AuthInfo aInfo = server.svr.whoAreYou(clientName)`

This method returns an `AuthInfo` object that includes the following elements:

- `serverName`
 - `clientName`
 - `timestamp`
 - a random number (nonce)
 - the keyed hash
2. The client verifies the `aInfo` object with the method `boolean verifies = aInfo.validate();`
If it is valid, the client creates its response `AuthInfo` object with the method `AuthInfo aInfoReply = aInfo.createReply(aInfo);`
The client then includes `aInfoReply` in the argument list for `createHandle`.
 3. The server verifies the `aInfoReply` object it receives by calling the object's `validate` method, and if it passes, the server continues to execute the rest of the function.

The use of the secret key, known only to this client and server, provides the data origin authentication guarantee, the timestamp and nonce provide for the guarantee of liveness (i.e. this is not a replayed message), and the cryptographic hash insures message integrity (and with the timestamp, loose sequence integrity).

Chapter 3

Enhancements to the India Protocols

My work is a game, a very serious game.

– M. C. Escher

The current version of the India protocols are not perfect; they can be attacked in a number of ways. This chapter discusses some enhancements to the protocols that can be used to defeat many of these attacks. At the end of the chapter, some attacks that do not have a known defense are discussed.

3.1 Assumptions

3.1.1 Tools

We assume the availability of several basic cryptographic tools, including a cryptographic fingerprinting function (also called a hashing function in many texts), a public key encryption system, a secure signature system (often implemented via a public key system, but not necessarily), and a secure random number generator. We do not specify how these tools are implemented.

3.2 The Write Proxy – Writing When Servers Are Down

One problem with our protocol is what to do when servers are disconnected (due to failures, network partitioning, etc) when a `putShare` is done. In the current protocol, the shares intended for those servers are lost. If enough shares are lost, then reconstruction may be difficult or impossible later.

One solution is to have a local write proxy take care of actually writing the shares and placing the shares immediately into a local read cache. In this manner, `putShares` can always return immediately, without any shares being lost, even if some servers are unreachable. When contact with those servers is reestablished at a later time (perhaps long after the client process has halted), the write proxy will push the shares out to it.

Note that even with a write proxy, the protocol still requires that at least one India server is reachable in order for the `createHandle` protocol to succeed. In theory, we could grant write proxy servers the authority to create their own writeHandles, but granting this level of trust to client hosts (which the servers implicitly distrust) would be a serious reduction in the security of our system as a whole. This means that disconnected operation is not possible for India clients (unlike Coda clients, for example), but we are willing to make this tradeoff in return for increased security.

3.3 Beating Sniffers – Secure Send and Deliver

The India protocols communicate over channels that may not be secure (and therefore must be assumed to be insecure). In order to defeat eavesdroppers, we can replace the standard “send” and “deliver” primitives with secure versions. (We do not propose to actually re-engineer RMI in order to do this, but simply to describe how this can be accomplished.)

Secure send and deliver require a public key encryption system and a secure random number generator. Since all messages sent as part of the India protocol are sent to a single recipient (i.e. there are no “broadcast” messages), each message can be encrypted by the public key of the intended recipient, so that only they can decrypt it.

3.3.1 Secure Send

1. Sender adds random “salt” to the message M , using a secure random number generator. The salted message is denoted M' .

The location of the salt in M' is defined as part of the protocol.¹

2. Sender encrypts M' by the recipient’s public key. The encrypted message is denoted M'' .
3. Sender sends M'' to the recipient as an ordinary message.

3.3.2 Secure Deliver

1. Recipient decrypts M'' salted message with its private key, giving M' .
2. Recipient removes the salt from the message, giving M .
3. Recipient delivers M as an ordinary message.

¹If a block cipher is used, then it must be scattered throughout the message, so that each block contains some random elements.

3.3.3 Properties of Secure Send and Deliver

1. Only the intended recipient can decode the message (if only the recipient has the recipient's private key).
2. Trolls that attempt matching-text ("dictionary") attacks (encrypting common or interesting messages with the recipient's public key, and then sniffing the wire for the encrypted forms of these messages) will be defeated by the random salt added to each message. As a corollary, successfully decrypted messages will not be recognized if they are repeated, because every time the same message is sent, its salted encrypted form is extremely likely to be different.

3.3.4 A Discussion of Public Keys

We assume that our public key system is secure and intractably hard to crack. If a troll manages to discover the private key of a client (by computational or other means), then it can impersonate the client and perform any operations that the client can. There is no defense in our protocol against a troll who has obtained a client's private key—such a troll can do anything that the client is authorized to do. However, a troll that has obtained a client's private key may choose to remain a passive listener anyway— if proper logs are kept of India activity, a troll who masquerades as a client may leave a trail back to themselves, but a troll who patiently listens may hear a great deal without revealing its presence.

Note that public key systems are typically computationally expensive. It may be more efficient for the communicating entities to generate a shared key for a private-key encryption system, and then use this key for all subsequent communication. Even with shared keys, it is still necessary to randomly salt the messages if trolls are able to deduce information from the pattern of repeated messages (even if they are unable to decipher the messages).

We do not address the issues of network traffic analysis (what an attacker might learn from noting who sends what to whom and when) though this may provide a wealth of information to the attentive packet-sniffing troll.

Secure send and deliver do not address the problem of replay attacks— in theory a troll that overhears the conversation between a client and a server may, in some systems, be able to "replay" the client side of this dialog when conversing with another server in order to authenticate itself as the original client. This problem is addressed by the use of sequence numbers in the messages.

3.4 Quotas and Restrictions – Beating the Hoggy Client

A single India client may clog the system and make writing impossible for other India clients by quickly creating myriads of objects as fast as it can, until either server storage space is exhausted or every conceivably useful object name has been taken.

The current protocols do not contain any defense against this form of attack. One solution, is to use the same methods used by traditional data storage systems: assign each client a storage quota. This quota can limit the number of objects that the client can create, the total size of these objects, or perhaps the set of names that the client is allowed to use (see the Hoggy Server, section 3.5, for another attack which can be defeated by constraining the namespace of the objects each client can create).

We do not propose at this time to add a protocol for the distribution of quotas among all the servers. The client/quota relation must be maintained as shared, global state, and as such is very difficult to keep consistent (particularly if the client quotas change over time). Unless this is managed completely by elves, we believe that this requires server-to-server communication, something that is currently absent from the India protocols, and which we desire to keep absent.

3.5 Fingerprinting and Signatures – Beating the Hoggy Server

A single trollish India server may cause a great deal of aggravation by the following attack: when asked to create a writeHandle, it can quickly impersonate a client and request writeHandles for an object with the same name from all of the other India servers (or send a message to a trollish client asking it to do so on its behalf).

From the client's perspective, the attempt to create the writeHandle may have succeeded or failed (depending on what the server claimed), but if it tries to use the writeHandle (or tries to get a writeHandle for the same object from any other server), it will discover that the object already exists and it cannot be created.

In the current India implementation, potential servers are polled in an arbitrary but fixed order by clients attempting to create writeHandles; the server with the lowest ID number will always be asked to create writeHandles first. In general, though, clients can and should poll servers in a random or even troll-adapting order, so that no particular server is more profitable for trolls to attack.

There exists a solution— if objects include as part of their name the name of the client that created them, then there is no possibility of a client competing with any entity other than themselves to create a particular object. For example, the namespace of India objects could be partitioned by client names in a manner similar to the way that user directories are separated in the UNIX file system: two clients might create objects with the same name in their own space, and yet these two objects will have unique (and easily distinguishable) global names.

Client requests to create handles or shares would be signed by the client (and subject to mutual authentication between the client and server), so that requests cannot be forged and so that the identity of the requesting client is known to the server.

Partitioning the namespace in this manner adds a very undesirable layer of complexity to the system. It puts an added burden on the clients, who must now know how to navigate a more complex namespace in order to find what they are looking for (in the general case, they not only need to know the name of the object, but the identity of the creator). Trolls can force the issue by creating objects

with identical names to valid objects, where the only distinguishing feature of the valid objects are the names of the creating clients. Since India objects are immutable, if clients names are added to names of objects, then client names must also be immutable.²

3.5.1 The Rude Server

A degenerate case of a Hoggy Server is a server that simply refuses to create a `writeHandle` at all. It may claim that the object already exists (when it does not), or it might simply fail to return a `writeHandle` that the client believes is valid.

A defense against this attack is that when the client is refused a `writeHandle` that it believes it should have gotten, it can ask for the `writeHandle` from other servers. If the server is not hoggy, then this request will eventually succeed if the object in fact does not already exist.

3.6 Fingerprinting the Data and the Shares

When a server gives a `WriteHandle` to a client, it delegates substantial authority to that client. Currently there is no way to make sure that this ability is not abused; a client can deviously write shares consistent with one value to half of the servers, and shares consistent with a second value to the other half. The value of the object seen by other clients would depend upon the set of servers that they accessed, and thus violate the principle that all India clients see the same value for all India objects.

One solution to this problem is to extend the `createHandle` protocol so that the handles created by a correct server cannot be trivially abused in this manner, or if they are, then this is recognized during reconstruction.

We can do this by fingerprinting each share, as well as the original data, with a cryptographic fingerprinting function, and store these fingerprints along with each share on each server. These fingerprints are signed by the server (even though the server never knows what the data actually is), so that the client cannot substitute different data without detection.

3.6.1 The Fingerprinted Writing Algorithm

The changes to the writing algorithm are described below:

- **`createHandle`**

1. Client computes the value of all shares S_i (*before* requesting a `writeHandle`).
2. Client computes the share check vector by taking a fingerprint of each share.

²You would not want a troll to keep changing its name in order to get more disk space, nor would you want to discover that your namespace was already used up by some client who no longer exists but whose ID you have inherited.

3. Client computes the fingerprint of the data itself.³
4. Client requests a writeHandle for the given object. The request includes both the share check vector and the fingerprint of the data itself.

We haven't explained why it is necessary to fingerprint both the shares and the data itself, and why both must be included in the request. These fingerprints will be essential in the reconstruction algorithm given in section 3.6.2.

5. Server performs the server side of the createHandle protocol. It adds the share check vector and the data fingerprint to the writeHandle, and then signs the entire writeHandle, and returns the result to Client.
6. Client confirms that the server has encoded the correct share check vector and data fingerprint in the writeHandle.

If the server has incorrectly constructed the writeHandle by using the wrong share check vector or data fingerprint, then the client will be unable to convince any servers of the authenticity of the writeHandle later. Therefore, the client should check the writeHandle for correctness immediately, in order to detect if the server is a troll. If so, then Client can abandon this writeHandle and begin anew with a different Server. (However, in such a case the client can still be denied write permission by a *Hoggy Server*— see 3.5 for more details.)

- **putShares**

As usual, the client disperses its shares by sending the appropriate shares and a copy of the writeHandle to each server. When each server receives a share, it computes the fingerprint of the share and compares the result with the value encoded in the writeHandle. If the two values are identical, then the server believes that the share is consistent and records the share and the fingerprint. Otherwise, the share is refused.

The correctness hinges on the fact that we believe that the probability of accidentally (or purposefully) discovering two objects that have the same fingerprint is exceedingly small. Therefore, if the fingerprints match, the share is very likely to be the same one that the client intended to be written.

If the client is intentionally writing bogus shares, then we cannot discover it during the write process. However, the fingerprinted Byzantine reading algorithm will uncover them, as shown in the next section.

³In theory, there are cryptographic fingerprint functions such that the fingerprint of each share is the same as a share of the fingerprint, or at least there exists a transformation from one to the other. (There are non-cryptographic fingerprint functions that have this property, but unfortunately we require a cryptographic fingerprint.) Given such a function, it may be unnecessary to fingerprint both the shares and the data itself.

3.6.2 The Fingerprinted Reading Algorithm

The fingerprinted reading algorithm is a refinement of the Byzantine Read algorithm. It uses the fingerprints computed in the fingerprinted writing algorithm to identify bad shares much more efficiently (and effectively) than the original Byzantine Read algorithm, however.⁴

Note that this algorithm also defends against a troll who is able to edit the shares in an India servers database— these shares are discarded before reconstruction.

The reading algorithm is as follows:

1. Eliminate any shares whose share fingerprints do not agree with their own value of their fingerprint. (It should be impossible to write such shares, given earlier enhancements to the protocol, but a troll may have flipped a bit somewhere during the read.)
2. For each remaining share S_i :
 - (a) Compute the fingerprint f_i of the share.
 - (b) If there exists any share S_j that lists a different fingerprint for S_i than f_i , discard both S_i and the first such S_j .

When we discard, we do not know which share is incorrect— either or both might be inconsistent. Determining which is the bad share is sometimes possible (by repeated reconstructions), but for the purpose of this protocol, we assume that the majority of shares are correct. Therefore, we don't worry about discarding some good shares as long as we also get rid of any bad shares; eventually all the remaining shares will be good.

When this iteration terminates, all remaining shares have validated, via their fingerprints, that the β vectors of the remaining other shares are what was originally written. Any reconstruction from the remaining shares should give the same value.

3. If there are fewer than N shares remaining, then search the discarded shares for matching fingerprint vectors (remember that we may have discarded valid shares in the previous step, because at that point we didn't know the bad from the good— but now that we know, the decision is easy), and add matching shares to the good set. Repeat until there are N good shares, or all discarded shares have been reexamined.
4. If there are N shares remaining, perform the reconstruction and check whether it matches the fingerprint of the data as a whole. If so, report success, otherwise report failure.

⁴The original “reconstruct and vote” algorithm requires combinatoric work and cannot handle many trolls. The fingerprinted reading algorithm requires polynomial work and can handle many more trolls. I will spare you the proof, which originally stretched on for a few pages. See me if you really want to know. -DJE

As long as a majority of the shares are good, and that majority contains N shares, reconstruction is possible.⁵ We believe that there exist other ways to analyze shares and patterns of reconstruction such that bad shares can be identified by fewer servers.⁶ Unfortunately, we do not know any such method at this time.

3.7 Concealing the Share Database

An interesting point of attack is the share database itself. If the private key of an India server is cracked, and a troll can get a copy of the complete share database, what can the troll learn?

From the basic properties of IDA itself, we know that complete reconstruction of the data is impossible with fewer than N shares. However, with even a few shares, the troll can narrow down the possibilities. All of the elements of a share's β vector have a relationship with the β vectors of other shares; knowing the β vector of one share reduces the space of possible reconstructions by the inverse of the size of the field used by the IDA. For example, in our reference implementation, we use a field with sixteen elements, and N is 2. Therefore, there are 256 possible reconstructions per unit of dispersal (which in our case, is one 8-bit byte). If we know one of the betas, however, we can narrow down the space of possible reconstructions to 16 possibilities. (Learning another β narrows down the space of possibilities by another factor of 16, giving a unique reconstruction— which of course is the entire point of IDA.)

If the troll has a small set of candidate reconstructions to choose from, narrowing down the set of possible reconstructions even slightly may be sufficient to allow the troll to make the correct choice.

However, there is a simple solution to this problem. We can adopt the method of secret sharing developed by Shamir, to the task of hiding the information in the share databases.

The Shamir method of secret sharing is computationally similar to IDA. The difference is that instead of using the original data as the coefficients of a polynomial, Shamir only uses one data element as a coefficient (typically the coefficient of the constant term). The rest of the coefficients are chosen randomly, by a secure random number generator.⁷

In essence, the writing client is keeping secrets from the India servers— the India servers cannot reveal the secrets to a troll, simply because the servers don't know what the secret is.

With a share created in this way, fewer than N shares cannot be used to determine *any information whatsoever* about the contents of the correct reconstruction. Learning the values of additional shares reduces the space of possible reconstructions, but not in any useful way; even with $N - 1$

⁵This result seems to violate the properties of Byzantine agreement, which generally requires more good shares. We are using the signature of the server that created the writeHandle as leverage to slightly “un-Byzantine” the problem, but as we'll see in section 3.8.2, this opens a security hole.

⁶If for no other reason than that Michael Rabin has suggested that such a method exists. Unfortunately, the method he knows of is currently unpublished, and he has not shared it with us.

⁷Note that an α of zero should never be used with the Shamir scheme, because when α is zero, the coefficient of the constant term of the polynomial is immediately revealed.

shares, the space of possible reconstructions still includes one possible reconstruction per field element, and the data coefficient (the only one not chosen randomly) can assume any of these values. All the troll can learn is a linear relationship between the random coefficients and the actual data, but the set of solutions to this relation is as large as the space of possible correct reconstructions.

The unfortunate side effect of the Shamir method is that shares become larger by a factor of N . One trade-off that can be made is to decrease the level of security by decreasing the number of coefficients that are chosen randomly. Other schemes can be used to add random bits to the data (at known locations) before performing the IDA calculation. In general, this technique reduces the amount of information that can be deduced from the shares in proportion to the total size of the original data versus the size of the resulting share. This gives the user the ability to make a trade-off between the size of the shares and the desired level of encryption.

Properly implemented, this technique has the benefit that no troll can reconstruct any part of any of the data from a single set of shares, or even several shares. We can further strengthen this scheme by encrypting the data in the storage by a secret key held by the server. If this step is performed, then the troll needs to get copies of N the data from servers and N secret keys in order to learn anything about the data. A daunting task, even for a troll.

3.8 Attacks With No Known Defense

3.8.1 Stolen Keys

A troll who has stolen the key of a client can impersonate that client at will. There is no defense against this, until and unless the key is somehow revoked⁸.

A troll who has stolen the key of a single server may impersonate that server at will as well. Again, the troll can cause some mischief, but except during the granting of writeHandles (as described in section 3.8.2), there are no real opportunities to do harm. The troll cannot change the contents of other servers, see more than one share of objects, or fool clients, since they can double-check this server against others. A troll needs to take control of a number of servers before being able to do much except authorize bad writes.

3.8.2 The Devious Server

The devious server is a variation of the devious client attack (see 3.6.1). It may be staged by a troll that has taken over a server and has client write permission, or by a troll server and troll client in collaboration.

A troll server can generate multiple writeHandles that map to the same longName but have different share fingerprints. These writeHandles, in the hands of a troll client, can be used to successfully write inconsistent sets of shares. When reconstruction of these shares is attempted, the

⁸This would be a function of the public-key infrastructure that is used.

clients may either fail to reconstruct or may reconstruct different values of the data (depending on which subset of shares they receive).

The protocol extensions that deal with the devious client (see section 3.6.1) will prevent “obviously” incorrect reconstructions. However, there may not be any possible correct reconstructions, or the value of the apparently correct reconstruction may vary depending which share servers are accessible at any given moment.

3.9 Conclusion

Men of sense often learn from their enemies. It is from their foes, not their friends, that cities learn the lesson of building high walls and ships of war...

– Aristophanes

With these enhancements, the India system becomes quite robust. Although we cannot claim to be experts in the area of cracking distributed systems, we have tried to imagine what such attacks would be like, and believe that the methods we present are sufficient to defeat nearly all of them (the devious server being the exception).

3.9.1 The Trusted Computing Base

At this point, the trusted computing base becomes surprisingly small, relative to other systems:

- All entities must trust the public key and DNS systems, and the identity-based access control mechanism.
- Clients must trust that the servers that they obtain writeHandles from will not attempt the devious server attack (see 3.8.2).

Beyond this, very little trust is necessary.

For reading, clients do not need to trust *all* of the servers, as long as they can trust the majority of them. For writing, clients need to trust that the server granting the writeHandle, and that the majority of the servers will correctly record their shares. In order to maintain the secrecy of their data, clients must also trust that N of the servers will not collaborate to reveal their data.

Servers do not need to trust the intent of clients; rogue clients can only clutter up their own namespace and use up their own quota (if the corresponding changes to the India system are made).

Reading clients do need to trust that the writers wrote something meaningful and consistent (but this is no different from any system). In some cases, inconsistent reconstructions can occur.

Communicating entities do not need to trust the communications channels; all messages are encrypted and cannot be decrypted by anyone other than the intended recipient, and with mutual authentication and signing, forged messages cannot occur. Dropped messages are detected by assuming δ -timeliness in the communication channel.

Chapter 4

Prototype Implementation

In an India system, data objects are simply arrays of bytes, which we'd like to store in a distributed fashion as *shares* on a bunch of servers. Our prototype implementation is written in Sun's Java language, and passes messages using Java's Remote Method Invocation (RMI) protocol. For simplicity, our India system is homogenous; each data object requires the same number of shares to reconstruct, uses the same irreducible polynomial, etc.

All of our documentation is online at <http://www.eecs.harvard.edu/~india/code-docs/> as `javadoc`; this chapter is meant as a kind of road map to the code base, describing its overall structure and pointing out places of interest.

The major pieces of the code are:

`india.ida`: An implementation of the Information Dispersal Algorithm for our particular configuration is contained in this package. The class `india.Ida` provides a front-end to the Algorithm that abstracts the process of dispersing and reconstructing entire data objects and shares.

`india.server`: Here we define the remote methods that may be invoked on servers by clients of the system. We also provide a somewhat abstract database in which servers can store their shares.

`india.client`: The various algorithms for reading and writing such as simple reading and writing, Byzantine reading, and Shamir reading and writing are implemented as subclasses of `IndiaClient`. This allows for sharing of basic services such as finding servers and the mutual authentication protocol.

`testbed`: The actual client and server applications are implemented in the package `testbed`, and are discussed in chapter 5. `iserv` is discussed as it is currently our only `IndiaServer` application.

4.1 Basic types

At first, the number of packages, classes, and methods can seem somewhat daunting. Here we discuss some of the classes fundamental to our implementation of the system, which are found in the package `india`.

First, for the sake of definition, note that the objects stored in the system are simply fixed-length arrays of bytes.

4.1.1 Share

A *Share*, class `india.Share`, contains all the information necessary to identify and use a piece of IDA-dispersed data. Here we store the alpha element, the beta vector, and the length of the original data.¹

4.1.2 Handle

An `india.Handle` contains the `shortName` and `longName` of an object (which are just `Strings`). A `Handle` also contains an `unused`² field that could contain creation information, etc., to help a client distinguish among objects in the case where there is more than one `longName` for a `shortName` (due to a `WriteHandle` race; see section 2.2).

`Handle.validate()` is called when we'd like to verify that the `longName` is well-formed; in our implementation, this means that it must contain only one `Handle.separator` character, which must have a non-empty set of characters on either side of it. The substring before the separator is the `shortName`; the whole string is the `longName`.

4.1.3 WriteHandle

This class (`india.WriteHandle`) contains both a `Handle`, and the bytes representing a signature of that `Handle` by a server, which authorize the `Client` to write an object with that name.

4.2 The IDA

The algorithm used to implement the Information Dispersal Algorithm is somewhat beyond the scope of this document; for a guide to the IDA, please refer to <http://www.eecs.harvard.edu/~india/ida.ps>.

All of the underlying finite-field manipulations are carried out by the classes in package `india.ida`; the classes `Field`, `Field16`, `Matrix`, and `Vec` are all concerned with these operations. `india.Ida`

¹Since the IDA may need to pad bytes, it is important to remember the original length of the data.

²Well, not entirely unused; see section 4.3.

serves as a front-end to the IDA for the rest of the system; it provides `disperse` and `reconstruct` methods that operate on entire Shares and data objects, rather than on individual field elements.

The remaining code in `india.ida` resulted from an effort to increase the performance of these methods over our original naive implementation; `IdaCursor` is tied closely to `india.Ida` in order to require as little computation as possible.

4.3 Server

The only requirement on an `IndiaServer` is that it behave as specified by the `india.server.IndiaServer` interface; in Java, this means that such a class must declare that it implements `india.server.IndiaServer`. This interface extends `Remote`, making it an RMI object that can be remotely referenced and have its methods invoked. The methods are fairly self-explanatory, and almost exactly reflect those in the protocol specification (Chapter 2).³

The details of Share storage are managed by an abstraction called a `ShareDb`, which supports many of the `IndiaServer` operations, but locally, on single shares of an object. This way, more clever implementations of the `ShareDb` can be easily layered under an `IndiaServer` without any significant changes. Our own implementation, `ShareDbFile`, is built on the Unix filesystem so that elves might have an easier time doing their jobs.

Our reference implementation of an `IndiaServer`, named `testbed.iserv`, is single-threaded and supports verbose logging so that we can verify, at a superficial level, that it is behaving correctly. Additionally, it goes through the motions of authenticating `WriteHandles`, even though the authentication codes live in a shared directory; our servers pretend not to know about the other keys (see section 2.3). In our initial implementation of the authentication protocol, servers use the `CreationInfo` field of a `Handle` to identify themselves as the signer, so that other entities know which key to authenticate with.

Of course, an `IndiaServer` need not use any of these classes; it need only implement the interface `IndiaServer` as described above.

4.4 Clients

One of the fundamental obstacles in the India system is that of naming the clients and servers, and discovering those names. Unfortunately, the standard RMI classes require that one know the name of the machine on which the desired object requires – not too useful in a dynamic, distributed system such as India.

³Except for the `ping` method, which exists to make sure the `IndiaServer` object is really bound to the server when the RMI Naming registry says it is. Due to the Java Virtual Machine's reluctance to finalize objects on exit, servers often stay bound after they have gone.

For our testbed system, we chose to specify the names of all the servers in a class file generated by a configuration script. Thanks to Java's dynamic class loading, only this one file need be recompiled to change a server name.

The implementations of the various client algorithms, such as Simple/Byzantine/Shamir read, and Simple/Shamir write, are grouped in the package `india.client`. To support the server-naming just mentioned, all our clients subclass `IndiaClient`, which supports this basic naming. The read algorithms, in turn, subclass `IndiaReader`, and likewise with `IndiaWriter`. We take advantage of this subclassing to write general test clients (`testbed.iget` and `testbed.iput`), which do not depend on a particular algorithm.

Again, this is simply how we structured our code; other valid clients could be implemented completely separately, as long as they know about the interface `IndiaServer` that they must talk to.

Chapter 5

Testbed Environment

5.1 Overview

This chapter describes the environment and conditions under which we tested our implementation of the India protocols.

The testbed consists of some clients that have been instrumented to exercise the protocols in various ways, a suite of scripts that implement several elves, and several scripts that combine these tools to build more complicated tests.

5.2 Testbed Clients

5.2.1 `iget`

`iget` is a wrapper around an `IndiaReader` object. Any of the four `IndiaReader` clients can be called from `iget` using optional command line flags. Four different `IndiaReader` classes have been implemented— the Simple, Byzantine, and Shamir reading protocols, and Pause, a special version of read that pauses between each step of the protocol (allowing trolls time to do their dirty work). The choice of read protocol can be specified by the user on the commandline; the default is Simple.

`iget` also provides the option of performing transfer rate time trials, and a verbose mode which logs many of the events for subsequent analysis. It also allows the user to explicitly specify which servers to use, (including troll servers) instead of searching for good servers itself.

5.2.2 `iput`

`iput` is a wrapper around an `IndiaWriter` object. Any of the three `IndiaWriter` clients can be called from `iput`, using optional command line flags. Three different `IndiaWriter` classes have

been implemented – Simple, Shamir. and Pause. As with `iget`, the choice of write protocol can be specified by the user on the command line; the default is Simple.

`input` also provides the option of performing transfer rate time trials, and a verbose mode which logs many of the events for subsequent analysis. It also allows the user to explicitly specify which servers to use, (including troll servers) instead of searching for good servers itself.

5.2.3 `ils`

`ils` is roughly analogous to the UNIX `ls` command. It provides information about the shares of an object on a particular server, or about all the objects stored on a server.

Strictly speaking, `ils` is not an India client. It converses with servers using undocumented (and not very reliable or secure) protocols. Nowhere in the current India protocols is there a way to ask a server to report information on *all* of the objects it knows about– the only thing that a client can ask for is information about a particular client (whose name the client must know ahead of time). Nevertheless, as we developed our testbed, we realized that this facility would be extremely useful during testing, so we added a trap door to the servers to allow `ils` to learn more about the contents of a share database than would otherwise be available to clients.

5.3 Testbed Servers

There are no testbed servers per se; the servers that we have implemented are ordinary India servers (at least as far as any India clients are concerned). However, they do have several features that make them useful as part of the testbed:

- They contain very little internal state; ordinary files are used to store the share database, and no information about the share database is cached.

Although this invokes a performance penalty, it makes testing and debugging of the protocols much easier. Using files to store information makes it possible to write other programs that view or manipulate the share database without the knowledge (or cooperation) of the server. For example, `elf-put` (see Section 5.4.3) was useful for populating a share database to test the implementation of the Read protocols before the Write protocols were implemented.

- They are peppered with diagnostic messages and logging facilities. Although ordinarily silent (except in moments of great stress), in verbose mode the servers print out and record quite a bit of useful information.

5.4 Elf Scripts

To learn how to be an elf, read `testbed.doc` (in the `testbed` directory of the India distribution). The elf scripts automate several aspects of setting up a testbed environment, including defining the

locations of servers and their public keys, and editing share databases.

5.4.1 elf-config

`elf-config` configures the India system by setting the hardcoded system variables such as the RMI port, the host RMI URL and by calling the `elf-svr-list` script.

5.4.2 elf-keyconf

`elf-keyconf` simulates our public-key infrastructure by generating DSA public/private key pairs for each specified client and server in the India universe, and then generating a shared secret key for each pairwise combination of clients and servers (to simulate the shared secrets that would be derived from Diffie-Hellman calculations in a production system.) The keys are stored as files in directories (one for each entity).

5.4.3 elf-put

`elf-put` has two purposes: it can create empty share databases, and it can populate share databases of all of the servers without the servers' active cooperation. This is useful for loading up the system with some known objects.

5.4.4 elf-rm

`elf-rm` deletes all of the shares associated with a given object. This is outside the India protocol (since shares are immutable and cannot ordinarily be removed), but it's useful to empty out the universe every once in a while and start anew.

5.4.5 elf-svr-list

`elf-svr-list` does the low-level work of creating server identities. This script should *never* be called from the command line (except perhaps by an elf with something special in mind, or a troll.) It is used by the `elf-config` program to create the global table that contains the identity (name and RMI URL) of each India server so the clients can contact the servers.

Note that India servers are not actually started by this script, and indeed may not actually ever run.

5.5 Test Scripts

5.5.1 test-speed

This script tests the raw throughput of India read and write clients, using simple read, Byzantine read, Shamir read, simple write, and Shamir write. Results obtained from running this program can be found in Section 6.1.

5.5.2 test-byz

This script tests the robustness of the India Byzantine read protocol. It creates a simple object and then trollishly fiddles with the contents of the shares for that object in each share database, attempting to cause reconstruction to fail.

5.5.3 troll-share

This script allows one to manipulate and corrupt (or `troll`) individual shares in varied and insidious ways:

Remove Delete a share from one server's share database.

Truncate Changes the length of the share body without adjusting the length field in the share header.

Diddle Replaces a chosen byte with a new, random value. This allows for corruption of either the object serialization header or the actual share information.

Restore Un-troll a share, returning it to its original value.

Chapter 6

Benchmark Results

6.1 Throughput

Throughput of the India system is tested via the `test-speed` script.

The following tables show the typical throughput speed (not counting the initialization of the client) for several different clients reading and writing a 100K file (in this case, `/bin/awk`). All speeds are measured in Kilobytes per second.

The Java code being benchmarked was compiled (without optimization) using JDK 1.1.1 from Sun, and executed in the 1.1.1 Java VM.

6.1.1 Running Locally

The following speed tests were performed with both the clients and servers running on `gagava`, a Sun Ultrasparc.

Protocol	Speed	Servers Used
Read (Simple)	70 K/s	Two servers
Read (Byzantine)	34 K/s	Four servers
Read (Shamir)	15 K/s	Four servers
Write (Simple)	16 K/s	Four servers
Write (Shamir)	7 K/s	Four servers

6.1.2 Running Remotely - Slow Client

The following speed tests were performed with the clients running on `rioja`, a Sun SS-20, and the India servers running on `gagava`, a Sun Ultrasparc.

Protocol	Speed	Servers Used
Read (Simple)	60 K/s	Two servers
Read (Byzantine)	30 K/s	Four servers
Read (Shamir)	13 K/s	Four servers
Write (Simple)	10 K/s	Four servers
Write (Shamir)	5 K/s	Four servers

6.1.3 Running Remotely - Fast Client

The following speed tests were performed with the clients running on `gagava`, a Sun Ultrasparc, and the India servers running on `rioja`, a Sun SS-20.

Protocol	Speed	Servers Used
Read (Simple)	65 K/s	Two servers
Read (Byzantine)	32 K/s	Four servers
Read (Shamir)	15 K/s	Four servers
Write (Simple)	15 K/s	Four servers
Write (Shamir)	7 K/s	Four servers

6.1.4 Remarks

In the India protocol, most of the computation is done by the client. As these benchmarks show, the penalty for accessing the data via remote India servers is not very high, and the throughput is not nearly that necessary to saturate the network. Therefore, we assume that on these machines, the India protocol is CPU-bound instead of network-bound.

Given this assumption, the penalty for having a slow CPU is not nearly as much as we expected. The CPU of `rioja` is a SS-20, which is allegedly about one-half the speed of a Sun Ultrasparc, the CPU of `gagava`. Therefore, since we believe that the India protocols are CPU-bound on the client side, a client on `rioja` should run much more slowly than one on `gagava`. Although there is a measurable speed difference, it is not as pronounced as we guessed it would be.

This was surprising enough that we decided to benchmark the two systems. CaffeineMark¹ is the only general Java benchmark that we are currently aware of, so we tried it.

CaffeineMark Benchmark	gagava Ultrasparc	rioja SS-20
Sieve	264	193
Loop	272	259
Logic	260	247
String	331	245
Method	301	152

¹<http://www.webfayre.com/pendragon/cm2/>

Regrettably, it is unclear exactly how these benchmarks should be interpreted— and after talking to Jim Waldo, we believe that these benchmarks measure the speed of the `Date` methods much more than anything else, at least under Java 1.1. Our interpretation is that the UltraSparc (gagava) is much faster at some operations such as string manipulation and method invocation, but not much faster in terms of arithmetic, logical, and looping operations. Since we believe that the bulk of the processing time used by these benchmarks is used in the IDA code, which is mostly logical and arithmetic operations, we don't see as much benefit from running on an of the UltraSparc as other applications might.

6.1.5 Compiled Java versus Interpreted Java

All of the benchmarks listed above are for interpreted Java. For the sake of comparison, we also benchmarked the core IDA section of the algorithm using a just-in-time compiler, running on a 150 Mhz Pentium laptop² and a 166 Mhz Pentium running interpreted Java. Unfortunately, we could not test the entire protocol back-to-back, because the computer that had the just-in-time compiler was not on the network.

	150 Mhz Pentium JIT Compiler	166 Mhz Pentium JDK 1.1 VM
16-way dispersal	30 K/s	7.7 K/s
4-way dispersal (est.)	120 K/s	30 K/s
2-way reconstruction	230 K/s	100 K/s

We spent a lot of time optimizing the IDA reconstruction code, so there probably isn't too much more speed that can be squeezed out of it. The dispersal code is not as optimized, but benefits much more from the JIT compiler.

6.2 Correctness

This section describes our methodology for testing the completeness and correctness of our implementation.

We define correct behavior to be the behavior specified by the India protocol specification. The basic approach is to enumerate all the points in implementation where error could possibly occur. Then, for each of these points, we again tried to enumerate all the possible errors. Next we developed testing scripts that allowed us to induce each of the errors. We applied this approach to the implementations of both the India read protocol and the India write protocol.

For this test, only race conditions and failures that disconnect clients from the servers such as link and crash failures were induced. The test environment consists of 5 servers (not necessarily

²Thanks to Jim Waldo for loaning us the machine.

running all the time) and as many clients as necessary for the test. In this case, five shares are made but only two are necessary for reconstruction.

In the implementation, clients poll servers for information starting at server number zero and try servers in increasing order until they receive a response or determine that all servers are down.

`IndiaReadPause`, described in section 5.2.1, was used to test the correctness of the read implementation. There are two particularly interesting points where errors could occur: after `findServers` but before `getHandles`, and after `getHandles` but before `getShares`. At both points one or more servers may be down or trollish. As a result, at the first point, an empty or incorrect list of Handles may be returned. At the second point, individual servers may not have the object represented by the Handle, may corrupt the Shares, or just may not return the Shares.

`IndiaWritePause`, described in section 5.2.2, was used to test the correctness of the write implementation. Like the read protocol, there are two particularly interesting points where errors could occur: after `findServers` but before `createHandle`, and after `createHandle` but before `putShares`. Again, at both points, servers that the client assumes are alive may die or turn trollish. At the first point, an incorrectly signed `writeHandle`, or no `writeHandle` may be returned. At the second point, individual servers may not receive the Shares or may corrupt them before they are stored.

`IndiaWritePause` was also used to induce a `writeHandle` race. This occurs when two clients receive `writeHandles` for an object with the same `shortName` from two different servers. In this case, both clients should be able to write to all the servers.

Chapter 7

Known Bugs

- *Robustness*

Various portions of the code are not bulletproof; for example, the directories that make up a `ShareDbFile` are not thoroughly checked to be sure that all the components of the path are correct, etc; nor is a directory properly created if needed. These are elfish things, and we assume that trolls won't alter such things, though they may muck with individual share files themselves.

- *Bail-outs*

There are still possible cases where clients or servers might exit rather than recover gracefully from an error.

- *Server Unbinding*

`iserv` “normally” exits by having the user press control-C (or `kill`'d in some manner); however, the Java VM won't finalize the server when this happens, and thus it stays bound in the registry. The current workaround is to ping servers immediately when we get a reference to them from a registry. If they don't answer, then they're not assumed to be alive (despite what the registry would have us believe).

- *Configuration Errors and Error Messages*

If there is an error in the configuration of an India system, the servers and/or clients usually perish with very mysterious error messages (or worse, uncaught exceptions).

- *Inability to Specify `longNames` During Reading*

Two or more objects with the same `shortName` may exist (if two clients attempt to create a handle with the same name at causally identical times). This is handled correctly by our protocols, but our implementation of the `read` method does not allow the caller to specify

which of the objects to read— one is arbitrarily chosen. Ideally, there would be a way for the caller to specify which of the objects to read, if there is more than one (by using the longName of the object, instead of the shortName). In fact, the methods that accomplish the low-level operations of fetching shares from servers *do* allow (or force) the caller to specify which object to fetch, but this functionality is not available at a higher level.

Chapter 8

Future Work

- Eliminate or work around all of our known bugs.
- Implement a secret key generation protocol for the authentication process.
- Implement the extensions to the protocols discussed in Chapter 3, particularly the write proxy, fingerprinting and secure messaging.
- Generalize the implementation to more flexible configuration parameters. For example, IDA-field size, Shamir-factor, etc.
- Develop a protocol for object deletion (and mutability) that does not require server-to-server communication or violate the semantics of the delete operation. We are not sure if this can be done, but it would be an extremely powerful result.
- Refine our analysis of the number of trolls that can be withstood without fingerprinted shares.

Appendix A

Related Work

The India System shares many properties with distributed file systems. Therefore, works pertaining to distributed file systems are interesting to us.

Perhaps the two most common distributed file systems are the Network File System (NFS) [3] (which is considered to be the distributed file system standard) and the Andrew File System (AFS) [3]. Like the India system, both of these systems employ a client-server model. However, both systems handle reliability differently than the India System. NFS has a stateless protocol, so no recovery occurs in the event of either a client or server crash. AFS provides reliability via R/O replication. Another common distributed file system is Remote File Sharing (RFS) [3]. RFS works only with System V and is therefore in limited use.

Another distributed file system is Coda [3]. Coda is a highly available file system that is optimized for availability and performance. It achieves availability through server replication and disconnected operation (e.g. a persistent cache is temporarily used as a replication site). Coda also maintains a high level of consistency, even through network partitions. In the case of a partition, Coda resolves most conflicts transparently, and a tool is provided to resolve conflicts beyond the capacity of the system.

A file system that has a writing protocol similar to the India System's is the Cedar File System (CFS) [3]. CFS is a highly available file system that guarantees consistency by using immutable writes. A difference between CFS and the India System is that CFS has centralized file naming. A CFS filename includes the name of a server, and is kept on each server. In the India System, the name of a file is not linked to a particular server and therefore naming is not centralized.

Although much work has been done in the area of distributed file systems and object storage systems, based on a review of relevant literature, we seem to be the only group exploring IDA as the underlying information *storage* mechanism for an efficient, highly available, fault tolerant, and secure data storage system.

However, IDA is used in several related contexts. The IDA algorithm has been used to implement a fragmentation-tolerant version of TCP called TCP Boston [2]. In [1] IDA is used to provide

fault-tolerance through redundancy for broadcast disks. Forward error correction, an alternative to packet retransmission in multicasting, uses IDA to add partial redundancy to the data stream, and is described in [4]. IDA is also used to reduce the number of retransmitted packets in Bulk Multicast Transport Protocol [6], and in fault-tolerant file transmission for rotator graphs [5]. A hardware implementation of IDA, Seth, is described at <http://www.cs.bu.edu/faculty/best/res/seth.txt>.

For more information on IDA itself, please refer to the original paper by Michael Rabin [7], or see <http://www.eecs.harvard.edu/~india/ida/>.

Bibliography

- [1] Azer Bestavros. AIDA-based real-time fault-tolerant broadcast disks In *Proceedings of RTAS '96: The 1996 IEEE Real-Time Technology and Applications Symposium*, Boston, MA, May 1996.
- [2] Azer Bestavros and Gitae Kim. *TCP Boston: A Fragmentation-tolerant TCP Protocol for ATM Networks*, Boston, MA, 1996.
- [3] Uwe M. Borghoff. *Catalogue of Distributed File/Operating Systems*, Munchen, FRG, 1996.
- [4] Bob Fischer. *Error Correcting Codes for Internet Multicast – A Feasibility Investigation*, Boston, MA, May 10, 1996.
- [5] Y. Igarashi, F. Bao, A. Mei, Y. Hamada. “Fault-tolerant File Transmission by Information Dispersal Algorithm in Rotator Graphs.” In *Proceedings of the 16th international conference*, Hong Kong, May 1996.
- [6] Robert Morris. “Bulk Multicast Transport Protocol.” In *IEEE Proceedings of INFOCOM '97*, Kobe, Japan, April, 1997.
- [7] Michael Rabin. “Efficient Dispersal of Information for Security, Load Balancing, and Fault Tolerance.” In *Journal of the ACM*, Volume 36, Number 2, 335-348), New York, NY, 1989
- [8] Bruce Schneier. *Applied Cryptography* New York, NY, 1996