Intrinsic References In Distributed Systems

Kave Eshghi
Software Technology Laboratory
HP Laboratories Palo Alto
HPL-2002-32
February 7th , 2002*

E-mail: kave@hpl.hp.com

distributed, storage, hash function

The notion of intrinsic references, i.e. references based on the hash digest of the referent, is introduced and contrasted with that of physical references, where the referent is defined relative to the state of a physical system. A retrieval mechanism using intrinsic references, the Elephant Store, is presented. The use of intrinsic references in hierarchical data structures is discussed, and the advantages regarding version management, consistency and distributed storage are argued. Since intrinsic references are immutable, it is not possible to manage change by modifying the referent. A mechanism for handling change, the variable store, is described. Alternative architectures for a number of applications, including a file system and an email system, are discussed.

1. Physical References

References are ubiquitous in computing. Memory addresses, URLs, file names are references. Every reference has a *referent*- the data that is referred to.

By the reference-referent relation we have in mind the relation between two pieces of data, where given the first piece of data (reference) it is possible to retrieve the other piece (referent) unambiguously, and in a way that was intended by the designer of the system.

Commonly, the relationship between reference and referent is defined relative to the state of some physical system. Memory addresses refer to the contents of a particular section of physical memory on a particular machine. URLs refer to the contents of a file on a given web server. File names refer to the contents of a particular section of a physical disk. Consequently, if the state of the physical system changes, the referent changes too.

We call these types of references, where the relationship between the reference and referent is defined by the state of a physical system, *physical references.*

Typically, physical references encode a *location* on the physical system where the referent is to be found. They are often hierarchical, reflecting the structure of hierarchically organized storage systems.

In a distributed world, using the state of a physical system to establish the relationship between a reference and its referent means that all access to the referent has to be via the physical system in question. This makes the physical system a bottleneck, and also a potential point of failure: if it loses its state, the information is gone.

Attempting to overcome these problems by duplicating the physical system creates consistency problem, because the state of all the duplicate physical systems must be kept in synchrony. The larger the physical system (data base, file system etc.,) and the greater the physical distance between the duplicates, the harder it is to keep them synchronized.

2. Intrinsic References

Data is immutable. The number 4 is data, it never changes to any other number. The string of ascii characters “to be or not to be, that is the question” is data, it does not change.

Data, which does not change, can be uniquely, succinctly and universally represented by its (cryptographic) hash digest[1]. Modulo the probabilistic collision free nature of the hash function, a hash value used as reference has the following properties:

- State independent: given the data block `S`, its reference is uniquely determined by the sequence of bits in `S`. This relationship only relies on the definition of the hash function, and not the state of any physical system.
- Unique: a given reference `R` can only refer to the input block `S` from which it was obtained by applying the hash function.

These two properties of hash based references ensure that these references are immutable in a strong sense: nobody can change the referent of a hash based reference. We call references based on hash values *intrinsic references* because they only depend on the data, and not on the state of any physical system.

Why is this a useful thing to do? Because the hash digest is a small, fixed size sequence of bytes that can for all intents and purposes stand for the data it refers to. The data can be of arbitrary size.

Of course, even with intrinsic references, we need a storage mechanism for retrieving the referent of a given reference. But the referent is not *defined* with respect to the state of the storage device. It is defined as the data which, when the hash function is applied to it, will yield the reference. Thus we can duplicate the storage device at will, without fearing inconsistency. Upon retrieval, we can check the referent for correctness using the hash function. Having duplicated the storage of referents, the problems of a centralized store go away, without creating the consistency problems associated with duplicating physically defined referents.

We call the storage and retrieval mechanism for the referents of intrinsic references the Elephant store, discussed next.

---

1. A hash function is a function from a sequence of bytes, `S`, to a short, fixed size sequence of bytes, `H`. We call S the input block and `H` the hash of `S`.

- A hash function is *Strongly Collision Resistant* if it is computationally infeasible to find two different input blocks which have the same hash.
- A hash function is *Weakly Collision Resistant* if for a given input block `S`, it is computationally infeasible to find another data block which has the same hash.
- A hash function is *Probabilistically Collision Resistant* if for a given input block `S1,` the probability that a randomly chosen input block `S2` will have the same hash as `S1` is extremely small.

Cryptographic quality hash functions, such as MD5 [8] and SHA-1 [4], have been developed for use in authentication, digital signature and other security applications. These functions are understood to be collision resistant in all of the above senses, although no proof exists and future research might find algorithms to generate collisions for these functions.

3. The Elephant Store

The Elephant Store is the name we have given to a distributed mechanism that allows the storage and retrieval of data blocks using intrinsic references (the hash digest of the data data block). A data block is simply a sequence of bytes; the store is completely oblivious to the semantics of the data it stores.

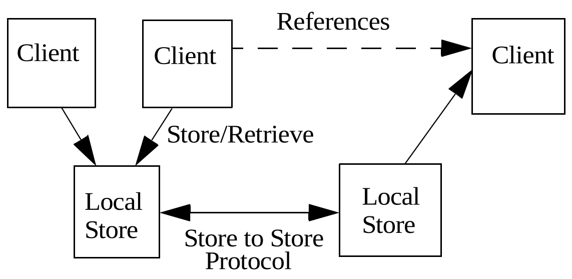


FIGURE 1. Architecture of the Elephant Store

T

he Elephant store is composed of a number of local stores. Clients connect to one or more local stores. Conceptually, the client API for the stores is very simple: it consists of the two operations `store(block)` and `retrieve(reference)`.

- `store(block)` stores the `block` of data in the store and returns a `reference`
- `retrieve(reference)` searches the store for the data block corresponding to `reference`. If it finds it, it returns the block, otherwise it fails.

Each local store keeps a table of `<reference, block>` pairs, indexed appropriately so that blocks can be retrieved given the digest. When a client issues a retrieve request, the local store first looks in its own table to see if the required entry is there. If it is, it returns the corresponding block to the client. If the entry is not in the local table, it tries to locate another local stores that might have the entry. If it finds such a store or stores, it contacts them to retrieve the block, and upon success, returns the block to the client.

The mechanism used for locating other stores that might have the required reference can be as flexible and heterogeneous as necessary. All that is required is to find a store that has the reference.

When the client requests to store a data block in the local store, the local store not only stores the data block in its own table, but informs the other stores (through the store to store protocol) that it has the corresponding reference.

Since the relationship between reference and referent is immutable, there is no way to update or manipulate data in the store, in the sense of changing the block referred to be a reference. Instead of updating existing data, the Elephant model only allows addition of new data. The fact that we may want to consider a new piece of data as a new version of an existing piece of data is purely up to the client, the Elephant store itself does not support any such notion. Since Elephant uses hash values as references, clients of the store can check if they receive the correct data from the store. The storage model removes any possibility of outdated (or stale) references. In fact, using hash values as references also implies that two identical pieces of data always will be referred to by two identical references, regardless of who initially created and stored the data, or where the data is stored and how it is retrieved. While the Elephant Store does not allow the modification of the reference->referent relationship, it is possible to remove data blocks from the Elephant Store, using a garbage collection algorithm. We will discuss this further in section 10.1.

4. Call by value semantics without the cost

From a programming point of view, parameter passing has had two possible semantics: value semantics and reference semantics. Value semantics copies the whole data structure across the (possibly remote) function call. Reference semantics copies a pointer (physical reference) to the data that is the value of the parameter. Value semantics is stateless, and greatly simplifies the design of distributed algorithms. Its use has been limited to small values (such as integers), because of the cost of communicating or copying large data values. Reference semantics is problematic from an algorithm design point of view, because the semantics of the reference rely on the state of a physical system (physical memory, or an equivalent surrogate), and can change (e.g. by the action of another process). Reference semantics has been used for all but the smallest data structures, due to the cost of copying or transmitting the data structure with each function call. Using intrinsic references allows us to have value semantics in function calls without the copying/transmission costs of reference semantics.

5. Storing data structures

So far we have treated data as a sequence of bytes, using the hash digest of the sequence of bytes as the reference. The next step is to build on this basic mechanism to extend the notion of data to complex data structures, particularly hierarchical data structures.

In order to apply the intrinsic reference mechanism to data structures, we need to map the data structure to a sequence of bytes, and later re-construct the data structure from that sequence of bytes. This is not a new idea in distributed computing, and all distributed programming platforms support serialization/deserialization or marshalling/unmarshalling.

In the context of intrinsic hash references, one point worth mentioning is that if we want to extend the identity property of intrinsic references (i.e. two references are equal if and only if their referents are equal) to data structures, then the serialization routine must be designed with this in mind, so that data structures that are considered semantically equal should always map to the same sequence of bytes. For example, two sets are equal iff they have the same members. If we want to detect the equality of two sets by comparing references to the sets, then the serialization routine for sets must ensure that equal sets are always serialized to the same byte sequence. We can achieve this, for example, by defining an order relation on the elements of the set, and making sure that the elements of the sets are serialized in ascending order.

5.1 Hierarchical Data Structures

The real power of intrinsic references becomes apparent with hierarchical data structures, i.e. when the referent itself contains references to other data structures. An example of such a data structure is the directory structure on a file system. Every directory is represented by a structure that includes references to the subdirectories and files it contains. All the references in this hierarchical structure, of course, are intrinsic or hash based references.

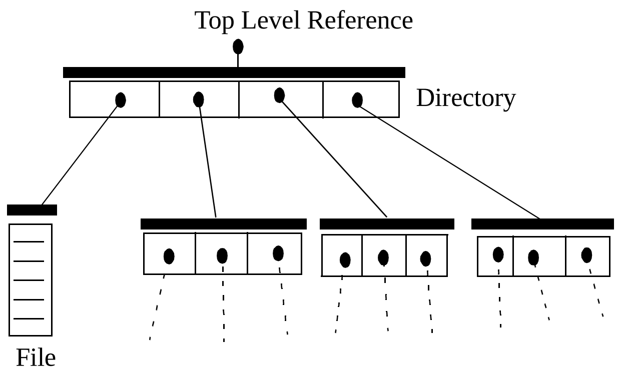


FIGURE 2. Elephant representation of directory structure

There are some points regarding hierarchical data structures that have important consequences for handling change. The first point is that two different trees can share a substantial number of subtrees. The second point is that the top level reference uniquely identifies the whole data structure, no matter how big and complex the lower level structures are. We will discuss the significance of these points in the section on change management.

6. Change

But what about change? If all references are immutable and intrinsic, how can we deal with change?

Physical references (such as memory addresses) have played a dual role in computing: they are used for denoting and retrieving the referent, and are also used for handling change by changing the state of the physical system that is used to define the referent. For example, a variable in a language such as C is really a surrogate for a memory location; we can use it to retrieve the referent of the variable, and also use assignment to change the referent.

When the data structure is large, change is often effected by modifying a part of the data structure, while keeping the rest of it intact. For example, if the data structure is an array, commonly only one part of it is modified by replacing one or more elements in the array without changing the rest of the elements.

This dual use of physical references is a source of many difficulties. For example, when concurrent processes can modify the referent of a physical reference, a great deal of care must be taken to ensure the integrity of the data structure and the other data structures that hold a reference to it.

We propose to deal with change using a dedicated mechanism that is separate from the reference-referent retrieval mechanism. We call this mechanism the variable store.

Simply put, the variable store is a table of associations between references. An association is a tuple `<V, R>` where `V` and `R` are intrinsic references. We call `V` the variable, and `R` the value of the `V`. A variable store is a set of associations such that for any variable `V` there is only one tuple `<V,R>` occurring in the set.

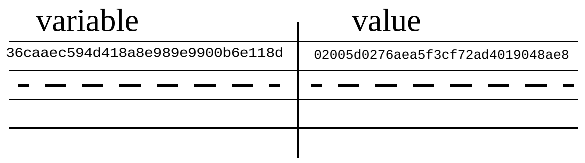

variable
value
36caaec594d418a8e989e9900b6e118d
02005d0276aea5f3cf72ad4019048ae8

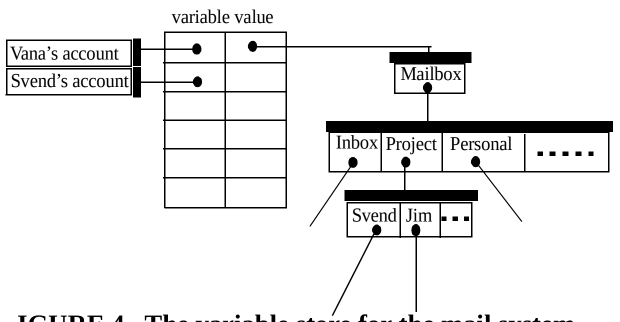
FIGURE 3. The Variable Store

The associations held in the variable store are not immutable; they can be changed by the clients of the variable store. Thus the variable store needs to be implemented using traditional mutable store technology, e.g. a database.

The state of the variable store reflects the state of the application. When the state of the application needs to be changed, the value of one or more variables are changed to references that represent the new state of the application.

It may seem that we have come back to square one on change management, since the mutable store used for keeping the variable-value associations is subject to the same limitations as traditional storage mechanisms such as data bases. But there is a crucial difference, and it concerns the granularity of the data structure that is modified when the state of the application is updated. We will illustrate this point using an example.

Consider a possible architecture for an email system using the Elephant store and the variable store. In this architecture, the state of the entire mailbox for each subscriber is represented by the value of a variable in the variable store. This value is the reference to the hierarchical structure representing the mailbox (Figure 4.)



IGURE 4. The variable store for the mail system

In the tree diagram on the right of the picture above, all the links in the tree are intrinsic references, using the hierarchical storage technique discussed previously. When a subscriber, such as Vana, receives a new mail, the mail server computes a new mail box structure from the old mailbox structure plus the just received mail, and replaces the old value of the Vana's mailbox variable with the new value. Figure 5 shows how the new mail box is computed from the old mail box and the new mail.

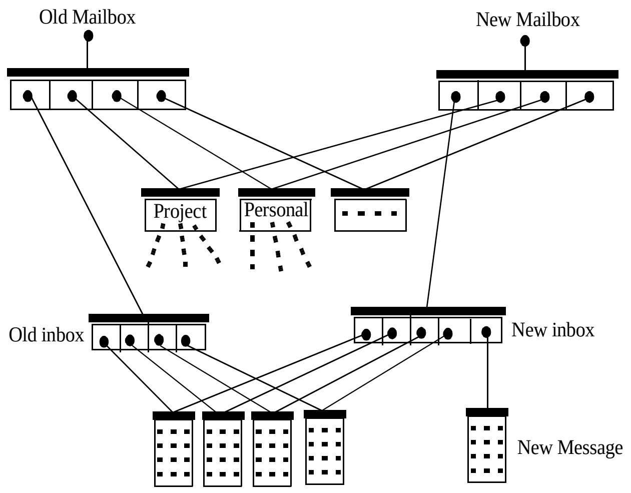


FIGURE 5. Adding a new mail to the inbox

The new mailbox and old mailbox data structures share all the data and references relating to the folders other than the inbox. In the data structure corresponding to the inbox, also, all the old messages and their references are re-used. Thus three new records are added to the Elephant store: that for the new message, that for the new inbox, and that for the new mailbox.

This example illustrates two crucial points about the way the variable store is used for representing state. The first point is that the variables in the variable store point to whole, internally consistent, data structures such as a mail box. At all the levels lower than that, immutable references are used. When there is a change, such as a new email in the inbox, we create a whole data structure to replace the old data structure. The second point is that by using hierarchical data structures, we avoid the cost of copying the bulk of the new data structure, since we can use structure sharing without loss of consistency.

As far as synchronizing her mail is concerned, all that Vana needs to do is to make sure that the value of the mailbox variable she stores on her email client is the same as the one stored in the variable store. Once this is achieved, for example by querying the variable store periodically, or when she wants to check her mail, she can have access to all the information in her mailbox from the Elephant store. If she runs a local Elephant store (as she should!) on her client machine, all the messages and data structures except the three mentioned above will be on her machine already.

Now consider the situation where Vana has two machines: one her desktop, and one a laptop. She can synchronize her mail on either machine using this simple procedure, and have a logically consistent mailbox accessible from both machines. Now contrast this with what happens with existing mail servers. She have a choice, either of keeping all her mail on the server, or keeping it on the client. If she keeps it all on the server, it helps keep it synchronized when she has multiple machines, but every time she wants to retrieve an email (even an old one) she has to get it from the server. If the server is remote (for example the laptop is connected through a modem) this involves considerable cost and delay. The other option, keeping it all on the client, avoids the cost of retrieving the mail from the server all the time, but makes it very hard to keep the laptop and desktop synchronized. Keeping the mail on both the server and client creates complex synchronizations problems.

Semantically speaking, when we copy the new value of the mailbox variable from the variable store, we are copying the whole mailbox, lock stock and barrel. Thus, from a semantic point of view, it is as if we keep the mail on the server, and every time we synchronize, we copy the whole mailbox. Physically, however, we only copy the reference, and when we retrieve the contents of the mailbox, only those bits of the mailbox that have changed.

Thus, even though we have to keep a centrally managed variable store, we still gain considerably by using the Elephant Store and variable store combination. This advantage is multiplied when, as is common, many mail messages in the mail system have the same attachment. The attachment will be stored in the Elephant store only once, and if it is already cached on a client machine, will not be retrieved again.

7. The client server architecture

In the client server architecture (and three tier architectures that are based on it) the server plays a multitude of roles. It is where the data is stored, where the state of the application is kept, where the application logic resides, and the logical entity that the outside world uses to access the service. We discuss these roles in more detail below, and then consider how these roles change using intrinsic references and variable stores.

7.1 Server as the provider of service identity

Consider the role of the email server on the network. It is the destination for all the incoming mail; the outside world uses the email address of the recipient to determine the server where the mail should be delivered. A similar role is played by a web server: it is the network location that URLs refer to. A file server, similarly, is a location on the network referred to by the name of the server. Thus the server name or network address acts as the server identity; the address `www.ebay.com,` for example, as well as providing a network address, identifies the service provided by EBay.

7.2 Server as the repository of data

In most traditional client server architectures, almost all application data is kept on the server. File servers and web servers are prime examples of this. With mail servers, as discussed previously, there is a choice to keep most of the data on the client, but the leads to problems with synchronization.

7.3 Server as the maintainer of the (consistency of) state

When the application has to deal with changes to the world, the state of the server determines the state of the application. We already discussed how the mail server keeps the state of mail box. In a web server, the coherence of the site is maintained by the state of the server, for example links within pages point to other pages within the site, which are kept consistent by the server.

7.4 Server as the seat of application logic

In client server applications, application logic is invariably executed on the server. This is largely a consequence of the fact that all the data is kept on the server, and it is more efficient to run the application logic close to where the data is.

8. Alternative architecture

The multiple roles played by the server in the client server architecture make it very difficult to distribute the functionality of the server. Thus in the client server architecture, while the clients can be distributed, the server is typically centralized. Using the Elephant store and the variable store as discussed previously, we can drastically reduce the role of the server, and avoid most of the problems associated with the centralization of the server.

We argue that providing service identity for the application is in essence the same problem as maintaining application state and providing access to this state using a well known name or identifier. When a variable store is used to maintain application state, service identity can be considered as the state of the variable store and a well known name that allows access to the variable store. Thus in our proposed architecture, the server is a mechanism that can be accessed using the service name and the existing network infrastructure, and where the integrity of the variable store is maintained.

Using the Elephant store, the storage of data need not be on the server. Data can be distributed and cached at will, without loss of consistency.

Once the storage of data is distributed, application logic can be distributed too. Application logic consists of two types of operation: computation on existing data to create new data, and, and updating the state of the application.

Computing new data from existing data can occur at any point where the input data and is available. For example, when a message is moved from one folder to another in the mailbox, this can done by the email client. The client computes the new mail box based on the old mailbox, and the given user action.

The second aspect of running application logic, i.e. updating the application state, boils down to updating the value of one or more variables in the variable store.

Thus the only role that remains for the server is to provide an access mechanism, based on a network name, for the variable store, and to provide access control mechanisms to ensure the integrity of the variable store on updates.

Let us look at an example other than the email system, for this proposed new architecture. Consider a news oriented web server, such as CNN or Yahoo. One possible way to implement such a site using the Elephant store and the variable store is to keep all the contents of all pages in the Elephant store, and use the variable store to associate file names (variables) with the Elephant reference to the contents of the file. In this architecture, the HTTP daemons are distributed and stateless. A client wishing to access a page in this architecture would go through the following steps:

- The client send its request to the HTTP daemon, specifying the file name it wishes to retrieve (using the URL, in the usual way)
- The HTTP daemon uses the first part of the URL (the domain name, e.g. www.yahoo.com) to locate the variable store. It uses the second part of the URL, the file path, to compute the variable that corresponds to this page. It does this by applying the hash algorithm to the file path. This makes sense, since the variables are the Elephant references of the file path. It then asks the variable store for the value of this variable.

- The value of the variable is the reference to the contents of the file. When the HTTP daemon receives the value of the variable, it retrieves the contents of the file from the Elephant store, and sends it back to the client.

When the contents of a file need to be changed (for example when the news is updated) the server adds the new content to the Elephant store, and updates the value of the variable associated with the updated file.

Thus the role of the ‘server’ is reduced to updating the Elephant store, updating the variable store, and providing an access mechanism to the variable store. The HTTP daemons, the ‘web servers’ of old, are not properly part of the server any more. They could in fact be located on the client machine, as proxies to enable standard web clients to work.

Notice that since all content is retrieved from the Elephant store, caching is automatic, based on the distributed nature of the Elephant store. Only requests for the value of variables need to go the ‘server’. Every request and response to the variable store is less than a hundred bytes, thus greatly reducing the bandwidth requirements to the (centralized, in this example) variable server.

9. Other applications

We have already mentioned two applications: email, and a distributed web server. Perhaps the most interesting applications of this idea are peer-to-peer. Here are a list of possibilities: each one needs to be elaborated further.

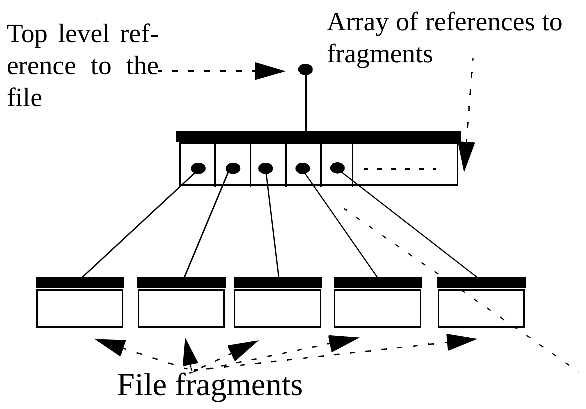

Top level reference to the file
Array of references to fragments
File fragments

FIGURE 6. Storing streaming files

1. Peer to peer streaming. In this application, multimedia files (video, audio) are broken to a number of fragments, each fragment relatively small (order of 10k bytes). Each fragment is stored in the (distributed, peer to peer) Elephant store, and the array of references is also stored in the elephant store (Figure 6). When a client wants to listen to a piece of music (say) she first gets the reference to the array from somewhere, maybe a search engine. Then she retrieves the array of references to the parts from the elephant store, and then starts retrieving each of the parts in order from the Elephant store. Since all access to the Elephant store is peer based, she would be able to retrieve the data from one of the peers close by, if they have it. At the same time, since the stream is broken into may small parts, people can start listening at different time without creating a consistency problem for reassembling the parts.
2. Distributed, versioned file system. The idea is to use the hierarchical scheme in Figure 2 to represent the structure of directories and files. All the contents of the files and directory structures are kept in the Elephant store. When a file is added to a directory, or an existing file is modified, we create a new tree (that shares most of the components of the old tree) and store it in the Elephant store. Thus every state of the file system is represented by a different reference. In one possible scheme, there is no central variable store. Each user stores the reference to the version of the file system he is using. If two users want to share a version of the file store, they make sure they are using the same top level reference, which they can do for example by sending each other email. The advantage of this file system is that a) it is distributed and duplicated, but consistent (for each user) and b) old versions and new versions can live side by side and be shared.
3. Decentralized discussion groups. The participants in the discussion add their messages to the data structure representing the history of the group, and send other participants the reference. All messages include references to the other messages that they are replying to. Thus the context of every message is clear. This is an example of a distributed application that does not need a variable store, except maybe to hold the email addresses of the participants.

10. Research questions

What are the major research questions raised by the type of approach discussed above? Here is a preliminary attempt at formulating these questions. The questions have been partitioned into the following segments: those relating to the Elephant store itself, those relating to the variable store, and those related to the architecture of applications built around Elephant and variable stores.

10.1 Questions relating to the Elephant store

1. Efficiency and practicality of using hash digests as references. Hash digests differ from physical references in that they don’t have any structure, while almost all physical references have a hierarchical structure that helps identify the location where the referent is stored. Thus we will have to find new and efficient mechanisms to locate Elephant stores that might have the referent for a reference that we are seeking. The scope for innovation here is very large, going from centralized search engine type mechanisms to gossip based peer to peer mechanisms. A

lot of the issues raised by the Elephant store are shared by the attempts to build a universal, non-location based file store [2][3][6] and similar solutions would apply.

2. Garbage collection. In the Elephant store we never change old data, just store new data. Thus there will be a point that we run out of space if we don't remove some of the old data. The garbage collection algorithm can be application dependent, or decided by the owner of each local Elephant store.
3. Network topology and storage policies. How should distributed Elephant stores be organized, how should data be deployed and migrated etc.

10.2 Questions relating to the variable store

The way the variable store was described above presents it as a centralized, conventional store. Certainly that is one way of implementing it. There are real advantages, and challenges, to distributing the variable store while keeping its integrity and service identity. This would be a rich area to do further research.

10.3 Application architecture questions

We have already sketched a few application architectures that would benefit from the ideas presented here. Some of the research question arising out of these examples are:

1. Using these ideas, to what extent is it possible to create a distributed, peer to peer architecture for applications that are traditionally considered client server applications.
2. Are there applications where application state can be fragmented, in such a way that the state information for one user (say) can be efficiently stored on the user's machine? We are used to thinking of application state as a monolithic construct kept in a central server. This assumption may not be true of all applications, and this has important consequences for how we implement the variable store.
3. How can we represent and execute application logic in a distributed world based on these ideas? What are the assumptions, and trade-offs?

11. Related Work

Using hash digests as a reference mechanism was first mentioned in the discussions on URI (Universal Resource Identifier) mechanisms. However, it was never adopted as a way to implement URIs.

In [5] a scheme is proposed for using hash digests as *content derived names* (CDN) for software configuration management. Under this scheme, software libraries would be identified using their CDN, and the programs that need to load them would use the CDN instead of the file name. This, it is argued, would overcome the version management and library incompatibility problems that occur with modularized software deployment.

Hash digests have been used extensively for file comparison, for example in [1], where it is used for avoiding the duplicate storage of identical files, and in backup systems. The use of digests for file comparison is different, however, from their use as a reference mechanism, used for retrieving documents.

There are many proposed systems for a location independent, distributed file system, such as PAST[3], OceanStore[6] and Freenet [2]. None of these systems use the hash of the contents of the file as the reference. The architecture and search mechanisms used by these systems, however, are relevant to the implementation of the Elephant store since they use opaque, location independent file identifiers.

12. References

1. W. J. Bolosky, S. Corbin, D. Goebel, J. R. Douceur, "Single Instance Storage in Windows® 2000", *Proceedings of the 4th USENIX Windows Systems Symposium*, 2000, pp. 13-24
2. I. Clarke, O. Sandberg, B. Wiley, and T.W. Hong, "Freenet: A Distributed Anonymous Information Storage and Retrieval System", *Designing Privacy Enhancing Technologies: International Workshop on Design Issues in Anonymity and Unobservability*, LNCS 2009, ed. by H. Federrath. Springer: New York (2001).
3. P. Druschel and A. Rowstron, *"PAST: A large-scale, persistent peer-to-peer storage utility"*, HotOS VIII, Schoss Elmau, Germany, May 2001.
4. FIPS 180-1, "Secure hash standard", *Federal Information Processing Standards Publication 180-1*, U.S. Department of Commerce/N.I.S.T., National Technical Information Service, Springfield, Virginia, April 17 1995
5. Jeffrey K. Hollingsworth, and Ethan L. Miller , "Using Content-Derived Names for Configuration Management", *997 ACM Symposium on Software Reusibility* (Boston, MA May 1997)
6. J. Kubiatowicz et. al. "OceanStore: An Architecture for Global-Scale Persistent Storage', *Proceedings of the Ninth international Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000)*, November 2000.
7. A. Menezes, P. van Oorschot, S. Vanstone, "Handbook of Applied Cryptography", CRC press, 1996, Section 9.4.2, pp. 349-351
8. R. L. Rivest, "The MD5 Message digest algorithm", *Request for Comments (RFC) 1320,* Internet Activities Board, Internet Privacy Task Force, April 1992