Global Identification of Storage Objects in Network Attached Storage

Iliya Georgiev, Kiril Boyanov, Ivo Georgiev

Abstract: *Network attached disk storage is an independent Internet node and provides geographical distribution, mutual identification and exchange of storage objects. The paper describes a method for global identification of storage objects and a structure of object registry. For the identification word-wide unique identifiers are generated by double hash transformation based on the Advanced Encryption Standard (Rijndael) with no collisions.*

Key words: *network attached storage, global identification of storage objects, cryptographic hashing*

INTRODUCTION

Disk storage architectures provided by the industry today are direct-attached storage (DAS), storage area networks (SAN) and network-attached storage devices (NAS) [3]. Direct-attached storage is connected directly to the I/O bus and offers high performance and minimal security concerns, but there are limits on connectivity. SAN is a part of a switched network that provides fast scalable interconnect for large numbers of independent computers and storage devices. Both DAS and SAN are block-based. The host operating system is responsible for mapping their data structures (files and directories) to blocks on the disk devices.

NAS provides file serving allowing data sharing across platforms and application servers. These servers (Web servers, e-commerce servers, bio-information repositories, digital libraries, multimedia servers and others) have different functionality and information exchange load with the storage.

The main service, which NAS devices offer to such servers, is geographically distributed repository of storage objects. In Internet-based global computing, it was observed that the servers are very powerful and fault-tolerant, but are still vulnerable to an unexpected catastrophe such as natural disaster or planned attack. All of the stored data could be destroyed instantly. In order to make mission-critical data operational even after a devastating strike, it is highly imperative to have distributed information and remote disaster recovery. The decentralized and dispersed nature of NAS gives the potential to be robust to faults or intentional attack, making it ideal for long-term repository.

Network attached storage has driven increasingly complex processing and optimization inside storage devices. Data stored in network storage is considered a collection of *storage objects*. Such collection can be mapped onto different file structures. Mutual identification and exchange of storage objects between different devices face problems of data distribution and security.

In this article, we address one of the research problems in distributed file service of NAS: *global identification of storage objects*. The described approach is implemented in a NAS under development.

FILE SERVICE IN NAS

Network attached storage exists in Internet environment. The provided file service depends on the storage usage and the demand for greater interactivity. The global computing requirements suggest two usage scenarios for the provided file service.

In the first one a server owns several geographically distributed NAS, connected via Internet. The NAS devices serve as a remote storage to the servers, but also can replicate data between each other, insuring consistent data copies. The NAS filer is responsible for mapping the application data structures to blocks on the storage devices. Additional metadata is transferred from the server to the storage to do such mappings. The metadata is controlled completely on the embedded NAS controller. In this scenario the NAS functionality offers file services compatible to the low-level file systems implemented in the most popular OS:

a. Usage of iSCSI that is an IP based storage-networking standard for linking data storage facilities. By carrying SCSI commands over IP networks, iSCSI is used to facilitate data transfers over intranets and to manage storage over long distances.

b. File sharing systems like MS Common Internet File System that runs over TCP/IP and utilizes the global domain name service. For any particular file, NAS determines the name of the server and the relative name within the server, exchanges messages with the server for connecting, opening a file, reading its data, closing the file and disconnecting from the server.

The second scenario is to incorporate a *distributed file service* in NAS, which requires algorithms for performing distributed search and related global identification. Such file sharing system enables to exchange storage objects directly among different NAS without the need of a central file server. The network-attached storage is used as a P2P (peer-to-peer) node. Distributed file service is a natural characteristic of such NAS, which extends the local computer storage with a pool of storage spread throughout the Internet.

METHOD FOR GLOBAL IDENTIFICATION

One of the challenges of designing NAS with a distributed file service, is how to find any given object in a large geographically dispersed storage. In classical file systems, the files are identified by a pair, where the first part is the identifier of the computer (usually the host IP address) and the second part is the path in the directory hierarchy. Such identification is not usable in NAS with a distributed file service, because the information can migrate over hosts with different IP addresses, and also the directory hierarchy is not enough informative when the stored data is considered as a collection of data objects. The approach accepted in our system is to assign a *storage object unique identifier (SOID)* to each storage object and to provide a *storage object registry* of the objects.

We propose each unique identifier to be generated by a very exacting algorithm that is sufficiently precise to prevent any two SOIDs from ever being generated in duplicate. Additionally a digest (search key) is extracted from the SOID. The digest supports fast lookup and searching of the corresponding storage object. The SOIDs are hash codes, generated by the proposed algorithm, which guarantee world wide consistent identification of such objects.

The purpose of object registries is to provide a storage object discovery platform for the Internet applications and an information framework for describing data objects exposed by any entity or application program. Using this framework, the registry manages the metadata about storage objects.

The process of data locating can be presented by the following paradigm: the application constructs messages referencing SOIDs and lets the NAS infrastructure pass these messages from peer to peer until the targets are located. Co-operation between the NAS and ability to route messages directly to objects without knowing their location are needed. Such functionality and interface require from the NAS to replicate, destroy, and migrate data to meet application-level goals [1].

In distributed file service, symmetric lookup algorithms could search the data where there is no a central node, and each node is typically involved in only a small fraction of the search paths in the system.

GENERATION OF SOID

The proposed method provides flexibility of use and identifier generation of different NAS entities. The NAS entity and file naming structure allows the establishment of unique names. There are two instances of names in the open distributed environment: externally visible and internal (globally unique).

The *externally visible (seen by humans) name* is referred to the storage object or NAS entity text name. This text name is usually of variable length, thereby making it internally difficult to use by the security facilities. In our method the text name is one-way hashed providing a visible name with fixed length and very high uniqueness.

The *internal name* is the *globally unique identifier (SOID)*, which also is a fixed length value. While the external names after initial hashing are unique enough to ensure that no security problems could result from duplication, the SOID is unique in both "time and space" ensuring that no two objects will ever be assigned the same value. In addition, it provides a basis for allowing object authentication to be performed once for an application session and for applications to access stored information in any NAS for which they are authorized. Data integrity and confidentiality services are enabled for use between any two nodes. Figure 1 represents the proposed method for SOID generation.

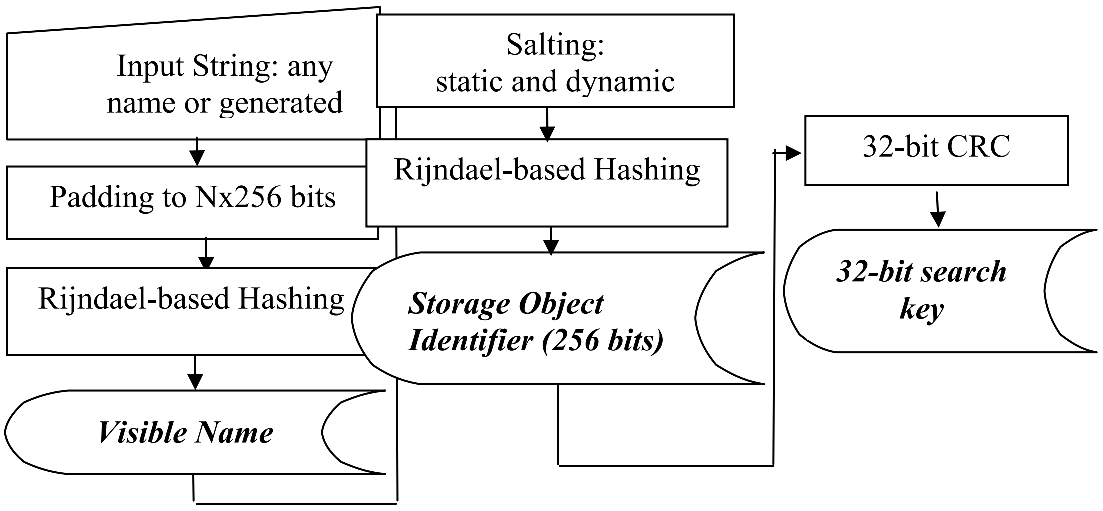

Figure 1. Generation of globally unique Storage Object Identifier

As *input string*, we use any text string, which could be derived from the object name or directory path attributes. If the object is just created and is considered anonymous, the input string is generated randomly.

The next step is *padding*. The input string is "padded" (by appending extra bits) as necessary to attain an overall bitlength, which is a multiple of the blocklength suitable for further processing with the following cryptographic hash function. First, the string is padded to make it a multiple of 256 bits long. Further we append a digit-one-byte, then as many zeros as necessary to make the string 32-bit multiple of 256, and finally a 32-bit representation of the length of the string before padding.

For visible name generation, the padded strings are further manipulated by a *one-way cryptographic hash function* that we have created and implemented using the fundamental principles of cryptographic hash functions [5]. It runs the basic Rijndael block cipher [2] that works with 256 bit key length and 256-bit block length and produces a 256 bits long external visible name. The one-way hash function H maps an input X of arbitrary finite bitlength, to an output $H(X)$ of fixed bitlength n. The hash input is divided into byte-length blocks X_i. The block cipher Rijndael performs simple whole-byte operations that make it very fast and appropriate for different platforms. Also, it provides extra flexibility, in that both the key size and the block size may be chosen to be any of 128, 192, or 256 bits.

In the proposed hashing the encrypted block and the key given to the Rijndael encryption are selected equal to 256 bits. Each block X_i serves as input to the internal fixed-size hash function, which computes a new intermediate result and the next input block X_i. Letting E_k denote encryption with key k, the block H_t is computed as follows:

$H_1 \leftarrow E_k(X_1);\ H_i \leftarrow E_k(H_{i-1} \oplus X_i),\ 2 < i \leq t.$

This is a cipher-block chaining, discarding ciphertext blocks $C_i = H_i$. The 256-bit padded string from the previous step serves as the key to encrypt a known plain text (we take it with all 0-bits). This yields a one-way function of the key (our padded string). The last cipher-text block is the hash value. The cipher-block chaining uses the previous hash value as the key (the very first time the previous hash value is equal to 32 bits zero), the message block (all 0-bits) as the input, and the current hash value EXOR-ed with the previous hash value and with the cipher output.

The generated visible name serves as input for the next step of our implementation - *salting*. To ensure fully theoretical and practical uniqueness the visible name is augmented (salted) with a string with higher degree of randomness. Salting is based on both static and dynamic attributes, which are taken in order to generate the SOID. The main goal is to choose among such attributes, which will yield always-different values for salting. The salt consists of two main parts - *static* and *dynamic*. The static part is a snapshot of certain system parameters at the moment the salting function was called for the first time. The static fields of the "salt" remain constant for a certain process, no matter how many times the salting function is executed. The parameters of the dynamic part of the "salt", in contrast to static fields, change every time the salting function is called. While dynamic parameters may happen to be the same on different systems or for different processes on a single system, it is very unlikely that the static fields would also match, i.e. two different systems were in the same state at a certain point in time.

We have chosen the following eight attributes for the static and dynamic parts of the "salt":

- **Static fields:** *PID* - process identifier; *System time* - system time taken at the start of this process; *Ticks* - the amount of CPU time since the beginning of the process (in *msec*); *Address* - a memory address taken during the first execution of the salting function;
- **Dynamic fields:** *System time, Ticks & Address* - same as the respective static fields but taken every time the function is called; *Counter* - a counter, which tells how many times the function was called (it insures against two calls of the salting function within less than a millisecond).

The salted strings are input again to a *one-way Rijndael hash function to produce the SOID.* Now the "salt" is the plain input text to be encrypted and the visible name is the key. The "salt" input is divided into byte-length blocks X_i, which are the chaining variables. The chaining variables go to the input and the visible name to the cipher key. The new value of the chaining variable is given by the old value EXOR-ed with the cipher output.

The next step is to produce a *32-bit Search Key*. The motivation to extract a 32-bit key for searching is obvious. Using a 256-bit SOID for searching is computationally expensive. A 32-bit digest ensures fast lookup in any data structure. For a digest, we decided to use a cyclic redundancy code with the generator-polynomial CRC-32, popular for the Ethernet [5]. The generator-polynomial is:

$$x^{32}+x^{26}+x^{23}+x^{22}+x^{16}+x^{12}+x^{11}+x^{10}+x^{8}+x^{7}+x^{5}+x^{4}+x^{2}+x^{1}+x^{0}$$

Using the excellent theoretical analysis in [4] we estimated the collision degree of such digest. This polynomial, when used with 256-bit SOIDs, is fully collision-free if the Hamming distance between the SOIDs is *1,2* or *3* bits. For *4* and more bits distance we used the formula $256!/r!\ (256-r)!$, where r is the distance in bits. The collision degree is slightly more than *1* out of 2^{32} generated search keys on average.

TEST RESULTS OF SOID GENERATION

For testing, we have produced a fully automated application. The test stand contains two applications. The first one is for creation of different random strings and for SOID generation. The second one is for analyzing of any possible duplicated search key from different SOIDs.

Some test results are shown into the following table below. There are no equal SOIDs generated – the double usage of Rijndael cipher makes the collision distance equal to 2^{256}. The only collisions we have are duplicated search keys from different SOIDs – in average one collision for 100,000 generations.

Number	*Type of string*	***Length of string** [bits]*	*Number of Generated strings*	*Collisions:* • NO for SOID; • Few for the search key.
1	Fixed	1	110,000	1 duplicated search key
2	Fixed	100	100,000	No collisions
3	Fixed	75	125,000	1 duplicated search key
4	Fixed	128	225,000	3 duplicated search keys
6	Sequence	32	100,000	No collisions
7	Sequence	100	120,000	2 duplicated search keys
8	Sequence	200	100,000	No collisions
9	Random	2	10,000	No collisions
10	Random	200	50,000	No collisions

STORAGE OBJECTS REGISTRY

Following the recommendations of the Internet society, we organize the discovery of the storage objects in *a two-level registry* for global identification.

The first level could be considered as a *table of the tuples "search key – SOID – Internet location (optional)"* of all consistent objects in the NAS. In a peer-to-peer lookup scenario such table is considered a *distributed search hash table*. The hash table abstraction provides a general-purpose interface for location-independent naming upon which a variety of applications can be built. The hash tables interface implements just one function: *lookup (SOID).* The function starts by performing a CRC to generate the search key. This key is used for fast search in order to check if the NAS has a tuple for the object. If such search matches, the NAS can retrieve the object or yield the network location currently responsible for the given SOID.

A distributed storage application might use this interface as follows. To publish a file under a particular name, the publisher would convert the name to a SOID, than call *lookup(SOID)*. The publisher would then send the file to be stored at the NAS, responsible for the SOID. A client wishing to read that file would later obtain its SOID, call *lookup(SOID),* and ask the resulting NAS for a copy of the file.

The second level of the registry represents *all known information* about a storage object. The individual instance data are sensitive to the parent/child relationships found in the application directory tree. Also some of the parameters could be changed during the live period of the storage objects like action enabled, permission bits and so on. The information associated with a storage object could be split into two groups:

a. *Identification information.* It is considered permanent and persistent during the lifetime of the data object. It contains the following data: *search key* (32 bits); *SOID* (256 bits); o*wnerName* - recorded name of the individual (computer, Web site, person, enterprise) that have created the object data. This is the certified name of the owner that manages the master copy of the storage object data and keeps registry information. The owner records this data at the time data is saved; *lookupURL* - a list of Uniform Resource Locators (URL) that points to file discovery mechanisms; *name* - required name recorded for the storage object; r*eferencedObjects* - a list of one or more logical or physical connected data objects; *description* - one or more short data descriptions.

b. *Parameters* that can be changed during the lifetime of the storage object: *functionsEnabled* - bits defining the operations performed on the storage object: open, edit, append, read, write; *permissions* - bits defining the actions allowed.

IMPLEMENTATION

Two different implementations of the method have been produced. The first implementation in C++ was oriented to verify and experimentally prove the method. The created automatic test system has been running for more than a month in order to collect information about the number of collisions. The input strings were generated with different lengths from one to 200 alphanumerical symbols.

The second implementation is on ARM symmetric multiprocessing architecture, using assembly language with no loop-carried dependencies and minimal cache misses. It is a part of the functionality of a NAS, oriented to provide distributed bio-information.

RELATED WORKS

The use of unique keys is a proved approach for identification of programming objects, digital documents and different computing resources. The Universally Unique Identifier is an identifier standard used in software construction, standardized by the Open Software Foundation (OSF) as part of the Distributed Computing Environment (DCE). The most widespread use of this standard is in Microsoft's GUIDs. The Microsoft algorithm has been justifiably criticized because of its collision degree and the usage of Ethernet MAC addresses [6].

We have carefully compared most of the generic methods on several criteria: collision ratio, portability to different platforms, NAS specifics, etc. After some analysis we decided to create a new method, which follows the paradigm of the standard but takes abvantages of double hashing function and is based on the last international cryptography standard.

CONCLUSIONS AND FUTURE WORK

An approach for global identification of storage objects in network-attached storage has been laid out. It offers a method for generation of globally unique storage object identifiers without collisions and a structure of an object registry. The method has been tested and implemented in a network-attached storage under development.

Our future work would be oriented to make our approach compatible with some of the emerging distributed environments like grid computing, Web services and P2P computing.

REFERENCES

[1] Balakrishnan. H., M. Frans Kaashoek, D.Karger, R.Morris, I.Stoika. Looking Up Data in P2P Systems. Communications of the ACM, vo.46, No.2, February 2003, pp.43-48.

[2] Daemen J., V. Rjimen. The Rijndael Block Cipher. Advanced Encryption Standard. At http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf.

[3] Gibson G. A., Van Meter R. Network Attached Storage Architecture. Communications of the ACM, Vol. 43, No. 11, (2000) pp. 37-45.

[4] Koopman P. 32-bit cyclic redundancy codes for Internet applications. Proceedings of the IEEE International Conference on Dependable Systems and Networks, 2002.

[5] Menezes A., van Oorshot, S.Vanstone. Handbook of Applied Cryptography. CRC Press, New York, 1997.

[6] Microsoft Globally Unique Identifier. http://en.wikipedia.org/wiki/Globally_Unique_Identifier

ABOUT THE AUTHORS

Iliya Georgiev, Prof. PhD, Department of Math and Computer Science, Metro State College of Denver, Campus box 38, P.O. Box 173362, Denver 80217, USA, Phone: +303 673 9403, E-mail: gueorgil@mscd.edu, URL: http://clem.mscd.edu/~gueorgil/

Kiril Boyanov, Acad. DSc, Central Laboratory for Parallel Processing, acad. G. Bonchev str.bl.25A, Sofia 1113, Bulgaria, Phone: +359-2-9796617, E-mail: boyanov@acad.bg

Ivo Georgiev, B.S., San Francisco State University, 1600 Holloway Avenue, San Francisco, CA 94132, USA, E-mail: ivogeorg@yahoo.com