The magic of content-addressable storage

Konrad Hinsen

The magic of content-addressable storage

Konrad Hinsen

The dry technical term "content-addressable storage" doesn't sound exciting, but superficial explanations can make it look like magic, to the point raising suspicion. In this article, I will show that content-addressable storage is a technology that works, is already in widespread use, and holds many promises for the future of both scientific programming and the management of scientific data. I will start by outlining the theory, and then illustrate how it works in practice, using IPFS, the Inter-Planetary FileSystem (http://ipfs.io/), as a vehicle.

1 A BIT OF THEORY

Traditionally, we use two techniques in computing to refer to data, which I will call *quoting* and *naming*. Quoting is just writing out the data explicitly, such as writing the number literal `42`, the text snippet `Computing in Science and Engineering`, or the few-thousand-byte-sequence consisting of the JPEG encoding of a photo, which I won't in fact quote here because it is too long. The last example illustrates the major limit of quoting: it is convenient only for small data. Naming works by defining a name in some namespace and *assigning* the data to it. Names, namespaces, and assignment take different forms in different technology layers. In a Python script, I can write `value = 42` to assign an integer value to the name *value* in the global namespace of the script. Modules, classes, and functions/methods are the other kinds of namespaces in Python. In my (Unix) computer's filesystem, the command `echo "Computing in Science and Engineering" > $HOME/my-preferred-magazine.txt` assigns a string to the name `my-preferred-magazine.txt` in the namespace of my home directory. In conventional jargon I'd say that I am writing a text string to a file, but semantically this is just the same as assigning a value to a variable, we just use different terms for historical reasons. On the Web, URLs are also names that live in namespaces that are managed by name servers (DNS) and Web servers. Assignment can take various forms, from copying a file to a server to on-line editing via a content-management system (CMS).

One reason for naming rather than quoting data is to define a shorthand for data that is too big to be quoted. Another equally important reason is that names are *mutable* references, i.e. the value assigned to a name can change over time. The URL for my blog is a name that remains constant in time, even though my blog changes as I add new posts and correct mistakes in older ones. A permanent reference to my blog is obviously useful. A third motivation for naming is that the name itself can be chosen to be descriptive. The URL of my blog is https://blog.khinsen.net, which is quite suggestive, at least for people familiar with Web habits.

However, the mutability of names is also a liability, in particular for names that are descriptive. Seeing the URL https://blog.khinsen.net, you may well suspect that it refers to my blog, all the more if you know that I use the handle `khinsen` on several social networks. However, you cannot really be sure of your assumption. Some evil hacker could have taken over my domain name and assigned different content to my blog's URL. While that's unlikely for my personal blog (it's not worth the effort), it is an issue for, say, my bank's Web server. Many risks in computing, be it Internet security or program correctness, are ultimately due to the impossibility to guarantee that the value assigned to a name is indeed what the manager of that name expects it to be. This is why functional programming [1] bans mutable names: a single name can refer to different values in different locations in the source code, but the value at each location is fixed over the duration of program execution.

The risks associated with mutability are probably the most important problematic aspect of naming, but not the only one. When naming is used to define a shorthand for data that is too big to quote, coming up with a name can be a chore, as anyone who has had to work with temporary files has probably experienced. Programmers have to decide where to put a temporary file, how to name it in such a way as to avoid collisions with all other programs that are also busy making up names, and to remove the name reliably when it is no longer needed. Speaking of collisions, they can also be a problem when it's humans that make up descriptive names and great minds happen to think alike. That's why we see fights over the right to certain domain names on the Internet, for example.

Since neither quoting nor naming is without problems, wouldn't it be nice to have another way to refer to data? That's the starting point of the development of content-addressable storage, which uses *hashes* as references in order to provide most of the advantages of quoting while keeping the size of the references constant, irrespectively of the size of the data itself. In IPFS, which I will explain in more detail later, you store data by passing it to the IPFS software, which gives you a hash in return. You can then transmit the hash to someone else, or archive it for later use by yourself. Whoever knows the hash can instruct the IPFS software to retrieve the corresponding original data.

A hash is to data what a fingerprint is to a person: a compact yet almost unique descriptor. Hashes are the result of computing a hash function whose input is a data stream, i.e. a sequence of bytes. A given hash function produces hashes of fixed size. For example, the popular (though by now a bit outdated) hash function called SHA1 produces 20-byte hashes, which are usually written as 40-digit hexadecimal numbers. There are Web sites such as http://www.sha1-online.com/ where you can paste in some data and get its hash computed with a click. You can also easily find implementations of the most popular hash functions in lots of programming languages, because hash functions are so widely used, in particular in cryptography.

Like fingerprints, hashes cannot be truly unique. There are only $2^{8\times 20} \approx 1.46 \times 10^{48}$ different SHA1 hash values, which is exactly the same as the number of possible contents of 20-byte files. But files can be much longer than 20 bytes, so there must be many files that share the same hash, a situation known as a *hash collision*. The conclusion seems to be that a hash cannot be sufficient as a reference to data. And yet, hash-based content-addressable storage is in daily use for various applications without any major trouble. Git uses content-addressable storage for managing data inside repositories. In spite of its enormous popularity, you don't hear people complaining about data loss. How is that possible?

A first observation is that most possible file contents are of no interest. Data that actually carries information is structured. Think of what you keep stored on your computer: text in English, source code in Python, addresses in vCard format, numerical data in IEEE floating-point formats, etc. The data formats impose many constraints on which byte sequences actually make sense. Additional informal constraints further reduce the number of potential meaningful files. Most syntactically correct Python programs are invalid for some other reason, and even though `QQQqqqqqq` is valid content for a name field in vCard, it is not very probable that someone actually has that name. SHA2-256, today's default hash function in IPFS, yields 256-bit hashes of which there are almost as many as the estimated number of atoms in the universe. It looks reasonable to assume that there are enough distinct hash values to uniquely label each file content that we might actually encounter in real life, if the hash function does a good job of assigning labels to contents. This has in fact been a very active field of research for a while. As I said, hash functions are very widely used, so finding good ones is an important job. There are two criteria that are particularly important in practice: even a small change in file contents should change its hash, because small changes are frequent, and it should be very hard (practically impossible) to construct a file for a given hash, because that would allow various forms of attack on hash-based systems. The main reason why I described SHA-1 as outdated above is that such an attack has been demonstrated recently [2].

There is a vast literature on the question that I discussed in the last paragraph, which I won't even try to summarize. But one aspect of this question needs to be emphasized: no matter how good your hash function, hash collisions are never impossible but merely very improbable. All reasoning about hashes is of a probabilistic nature. For people approaching computing from a mathematics background, this is often hard to swallow. You want to prove that your software works, not merely argue that it works most of the time! When coming from a physics or engineering background, probabilistic reasoning is more familiar. In fact, quantum mechanics says that all of nature is ultimately probabilistic. It is not impossible for me to fall through the chair I am sitting on, the tunnel effect would in principle permit it. But it's so enormously improbable that I don't lose sleep over the risk. Transferred to computing, which is done by physical machines, if a piece of software has a much lower risk of failure than the hardware it runs on, it is good enough in practice.

2 FROM THEORY TO PRACTICE: IPFS

The Inter-Planetary File System (IPFS, https://ipfs.io/) is an ambitious research and development project aiming at reinventing the World Wide Web with many fundamental improvements. One major difference to today's Web infrastructure is the use of content-based addressing as opposed to location-based URLs. This is the only aspect I will discuss here, please refer to the IPFS documentation for any other questions you might have. Before going on, let me stress that IPFS should at this time be considered an experiment. In my experience it is reliable though a bit slow, but your mileage may vary. Relying on IPFS as a data storage medium is probably not a good idea for now.

If you want to play with IPFS yourself, you should start by installing IPFS software on your computer. The most convenient entry point is IPFS Desktop (https://github.com/ipfs-shipyard/ipfs-desktop), but you can find other options at https://ipfs.io/. There are IPFS bindings for many programming languages, which are listed at https://github.com/ipfs/ipfs. They are all rather light-weight, as they connect to an HTTP server run by IPFS Desktop, which does all the heavy lifting. In the following I will use Pharo (https://pharo.org/), a modern Smalltalk derivate, because it is a particularly nice environment for explorative programming. In fact, the IPFS bindings for Pharo (https://github.com/khinsen/ipfs-pharo) are a byproduct of my own experiments with IPFS.

Let's start simple: I will store the list of numbers `[1, 2, 3]` in IPFS:

```
#(1 2 3) storeInIpfs
```

Pharo prints the result as `IpfsCid(bafyreib5ragfuond76dbywu7yubse73goit3h6v4dwf6tti2wfzdajo5py)` The long string in parentheses is the IPFS content identifier (CID for short) for my list. An IPFS CID is essentially a hash, but it contains some additional information. The initial 'b' indicates that the CID has been encoded as a number in base 32. This is done to make sure it will survive all kinds of Internet transmission channels that might strip spaces and control characters or convert everything to upper case. Base 32 uses only the 32 characters `abcdefghijklmnopqrstuvwxyz234567/` and ignores case. Decoding the base 32 string yields the real CID, a sequence of 36 byte which I'll show in hexadecimal notation:

```
#(1 2 3) storeInIpfs hex
```

yielding:

```
017112203d880c5a39a3ff861c5a9fc503227f667227b3fabc1d8be9cd1ab1723025dd7e
```

The first byte is a version number, whose presence allows future extensions to the CID format without invalidating existing CIDs. The second byte indicates how the data itself is encoded. In our case, the encoding is CBOR, the Concise Binary Object Representation (https://cbor.io/), which is best summarized as a binary version of the popular JSON format. The choice of CBOR was ultimately made by my Pharo bindings, but it's today's standard encoding for linked data in IPFS. The third byte indicates the hash function, SHA2-256, and the fourth byte is the length of the hash, 32 bytes. The 32 remaining bytes are just that hash. IPFS can accommodate many hash functions, but SHA2-256 is today's standard.

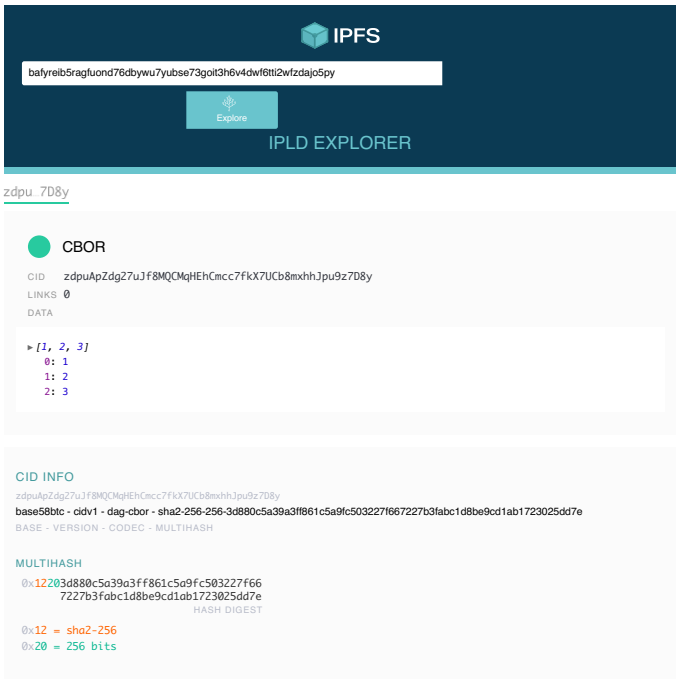

Fig. 1. My list of numbers in the IPLD explorer

You can now go to https://explore.ipld.io, paste this base 32 string into the search field at the top, and click "explore". You should see something very similar to Fig. 1. Under "Data" you see the list, which shows that the data transfer from my machine to yours has worked fine. In the "CID Info" box you see a confirmation of what I explained above. Except that the CID is printed as `zdpuApZdg27uJf8MQCMqHEhCmcc7fkX7UCb8mxhhJpu9z7D8y` This is just another encoding, the initial 'z' indicating base 58 instead of base 32. This leads to a more compact but also more fragile representation, as upper and lower case letters are no longer equivalent. IPFS software accepts any of these encodings, but recommends the use of base 32.

If you have done this little exercise, you have a first-hand proof that you can retrieve on your computer data that I have stored on mine, using nothing but the CID which is essentially a hash of the data. You may have wondered where the data is actually stored, and how IPFS manages to find it. I won't say much about the second part, because that would require me to explain IPFS' sophisticated peer-to-peer communication system. The basic idea is to send out requests to other nodes in the network, and choosing the nodes to turn to as a function of the CID itself. As for the storage location, there may be many but there is one I can be sure of: the IPFS pinning service at `https://temporal.cloud/`, where I pinned this CID. Pinning is IPFS jargon for instructing a server to keep a piece of data unconditionally. Otherwise, the IPFS software treats its storage area as a cache, from which items are deleted after a while if they are not requested. You can pin data on any computer that runs IPFS software. Pinning data on your desktop or laptop machine guarantees you an off-line copy, whereas pinning data on a computer with a permanent network connection effectively turns it into a server hosting the data. Pinning services are the IPFS equivalent of file hosting services, with a crucial difference: nobody needs to know on which pinning service you keep your data, because data requests involve only the CID, not the location. And that means that you are free to move to a different pinning service at any time.

Note also that all data stored in IPFS should be considered public. Even if you don't give your CIDs to anyone else, current or future versions of the IPFS distribution algorithms may send the data to other nodes even without an explicit request. If privacy is an issue, encrypt your data or use a dedicated network [3].

Location-independent data access is one of the big advantages of content-addressable storage. Another one is protection against accidental or malicious modification of data. Unlike traditional file systems, IPFS has no concept of write permissions, because it doesn't need it. There is no way to overwrite or replace any data because CIDs are not mutable names, but identifiers computed from the data. When you ask IPFS to retrieve the data for a given CID, you can recompute the hash to check that the data is indeed what you requested:

```
cid := IpfsCid fromString:
  'bafyreib5ragfuond76dbywu7yubse',
  '73goit3h6v4dwf6tti2wfzdajo5py'.
data := cid loadRaw.
hash := SHA256 hashMessage: data.
(cid last: 32) = hash
```

(The CID string is split into two parts to make the code fit into one column.) Pharo says `true`, so I know that I received the data exactly as its creator stored it. Note that `loadRaw` accesses the data at the lowest level: a sequence of bytes, which in this case stores the CBOR encoding of our three-element list. Hashes are computed at that level because that's how hash functions are defined. For most purposes, it is more convenient to use the two higher-level layers: the CBOR-encoding layer that I used above, or the file-system layer that most closely resembles traditional file systems.

3 LINKED DATA

There is one more feature of IPFS that I will explain in some detail, because it showcases another nice property of content addressable storage: the use of CIDs in data structures

to create immutable linked data, as defined by the InterPlanetary Linked Data specifications (see https://ipld.io). The idea is very simple: you can put CIDs into the lists or dictionaries you store in IPFS, and IPFS provides support for managing the resulting linked data structures and to access their elements. Let's see how this works in practice:

```
cidForData := #(1 2 3) storeInIpfs.
cidForDescription :=
  'List of three numbers' storeInIpfs.
(NeoJSONObject
  with: #data->cidForData
  with: #description->cidForDescription)
    storeInIpfs
```

In this piece of code, `NeoJSONObject` is a Pharo implementation of JSON objects, from a library called NeoJSON. The result as shown by Pharo is `IpfsCid(bafyreichcsaqqhxwgdpqvlni4p2z75arewt3yoesnvizgfxqkhiisbc52u)`, and by now you should know how to look it up using the explorer at https://explore.ipld.io. Please do so! The explorer will show a dictionary with the two keys `data` and `description`, each of which has a CID as its value. And you can click on the CIDs to inspect the data they refer to.

Now paste the string

`bafyreichcsaqqhxwgdpqvlni4p2z75arewt3yoesnvizgfxqkhiisbc52u/data`

into the explorer. As you will see, this will take you directly to the list. This notation is called a *path*, and most IPFS-based software will accept a path wherever it requires a CID. Paths work much like file paths on today's operating systems, except that they traverse dictionary-like data structures rather than directories.

It is no coincidence that my small example looks like data plus metadata. When data is stored in files, metadata requires an unpleasant trade-off. You can store the metadata along with the data in a single file, as it is habitually done in HDF5 [5], for example. But if the metadata contains execution-specific information, such as a time stamp or the name of the computer that did a calculation, the whole file becomes irreproducible because of the irreproducible metadata. Checking if two output files are equivalent in spite of the differing metadata is then laborious because it requires a detailed knowledge of the data format. Storing metadata in a separate file avoids such problems, and the raw data can be compared using generic tools such as Unix' `cmp`. But the link between data and metadata has to be made through filenames, and is therefore fragile when the data is transferred to a different computer where files live in a different namespace. Moreover, files being mutable, data-metadata associations are fragile even on the computer where they were established. With content-addressable storage, this problem disappears. In fact, checking if two computations produce the same result, assuming it is structured like my small example, requires no more effort than comparing the CIDs stored under `data`.

IPFS also helps with data management for linked data structures. As I mentioned, the elementary operation in data management is pinning, which means telling a specific node in the IPFS network to keep a piece of data indefinitely. IPFS can pin data *recursively*, which means that it also pins the data items referenced by any CIDs in the originally pinned data structure. A complex data assembly can thus be treated as a whole while still having well-identified substructures that can be shared with other data structures.

APPLICATIONS IN SCIENTIFIC COMPUTING

Let's assume we had a high-performance and widely adopted infrastructure for content-addressable storage. IPFS isn't quite there yet, but that could well change in a few years. How could we put it to good use for managing scientific data?

The most obvious use case is referring to data, including software source code, in an unambiguous and future-proof way. A journal article could simply state "the raw dataset has the CID ..." and both authors and readers could be sure to retrieve exactly the right data via this CID. URLs cannot guarantee the same level of permanence, nor can DOIs, whose permanence is no more than a moral obligation for the organizations that issue them. CIDs are therefore much more robust references for archiving, which is the reason why hash-based IDs very similar to IPFS CIDs are used in Software Heritage's archive of software source code [6].

Applied to software, content-addressable storage is the best approach we have today for ensuring reproducibility. The fundamental cause of irreproducibility is the complexity of the typical software assemblies we use in scientific computing. Both dependencies (mainly libraries) used by the software and build tools (compilers etc.) have an impact of a computation's result, and must therefore be recorded precisely. Storing all the files in IPFS and using the linked data approach I outlined in the preceding section would provide a permanent and unalterable record for future reconstruction. In fact, the package managers Nix and Guix use a very similar approach.

Pushing this idea further, we could combine the software dependency graph with a computational workflow graph and have a complete provenance graph for a computation, with everything stored as linked data in IPFS or an equivalent system. Yet another graph layer could provide the full history of the evolution of data and software. Since such graphs can become very large, and reference many and potentially large datasets, another welcome feature of content-addressable storage is the built-in deduplication: if you store the same data ten times in IPFS, only one copy will be kept, whereas a typical computer's file system contains many files with identical contents.

Another use case is the simplification of data communication between the different pieces of software used in a research project, e.g. the components of a workflow, or data processing software connected to visualization software. If all programs read and write data from and to content-addressable storage, communication only involves CIDs. Distributed computations does not require any additional effort because CIDs, unlike file names, are not specific to one computer. Simplifying communication between programs is a big deal because it encourages the use of smaller tools that do only one thing and do it well, a principle often called the "Unix philosophy". Smaller tools are easier to understand and easier to maintain, and that's something most computational scientists appreciate very much.

Finally, the transition from files to linked data structures should bring us further simplification. Using immutable references, we can package data in smaller units. Where today we use file formats such as tar, zip, netCDF, or HDF5 to ensure the coherence of multiple datasets and metadata, we will instead use simple linked data structures that don't require elaborate libraries to be integrated in every single piece of software.

In the long run, content-addressable storage could thus contribute to better computational science. Isn't that just the kind of magic that scientific research could profit from?

REFERENCES

[1] K. Hinsen, *The promises of functional programming*, Computing in Science & Engineering **11**, 86 (2009)
[2] M. Stevens, E. Bursztein, P. Karpman, A. Albertini, Y. Markov, A. Petit Bianco, C. Baisse, *Announcing the first SHA1 collision*, Google Security Blog, February 2017
[3] M. Ober, Dedicated IPFS Networks, https://medium.com/pinata/dedicated-ipfs-networks-c692d53f938d
[4] Enable IPFS Applications Through Private Networks, https://medium.com/temporal-cloud/enable-ipfs-applications-through-private-networks-28f98ea7358f
[5] Q. Koziol and D. Robinson, HDF5, DOI:10.11578/dc.20180330.1
[6] R. Di Cosmo, M. Gruenpeter, and S. Zacchiroli, Identifiers for Digital Objects: The Case of Software Source Code Preservation. iPRES 2018 - 15th International Conference on Digital Preservation, Sep 2018, Boston, United States. DOI:10.17605/OSF.IO/KDE56

Konrad Hinsen is a researcher at the Centre de Biophysique Moléculaire in Orléans and at the Synchrotron SOLEIL in Saint Aubin. His research interests include protein structure and dynamics and scientific computing. Hinsen has a PhD in theoretical physics from RWTH Aachen University. Contact him at konrad.hinsen@cnrs.fr.