YOUR CLOUD STORAGE PROVIDER DOESN'T NEED TO SEE YOUR DATA

Brian Warner
Zooko Wilcox-O'Hearn

Audience Participation Demo:
http://tahoe-lafs.org/RSA

zooko
starts

Who We Are

- Brian Warner

- Zooko Wilcox-O'Hearn

- developers of Tahoe-LAFS
- `http://allmydata.org/trac/tahoe`

What We’re Here To Talk About

Security of Data in the Cloud

Bring Your Own Security

Tahoe-LAFS

The Principle Of Least Authority

- Security of Data Stored in a Cloud
- Your Right to Security **and** Cloud Storage
- Better Options: Bring Your Own Security
- How Tahoe-LAFS Implements Those Options

What We Want You To Take Home

- Beliefs:
  - You deserve **provider-independent security**
  - Usability Matters!
  - The techniques in Tahoe-LAFS can be applied to other systems
- Skills:
  - Reliance Analysis
- Tools:
  - Erasure coding
  - Storing encryption keys in filehandles (capability-based security)

- You deserve confidentiality and integrity even when you buy reliability and availability from a cloud storage provider
- simplifies key management
- Tahoe-LAFS is an open-source system which offers easy-to-use

- Learn to identify which properties rely upon which components
- Erasure coding: provides tunable reliability-vs-overhead
-

Security of Data in the Cloud

brian takes over

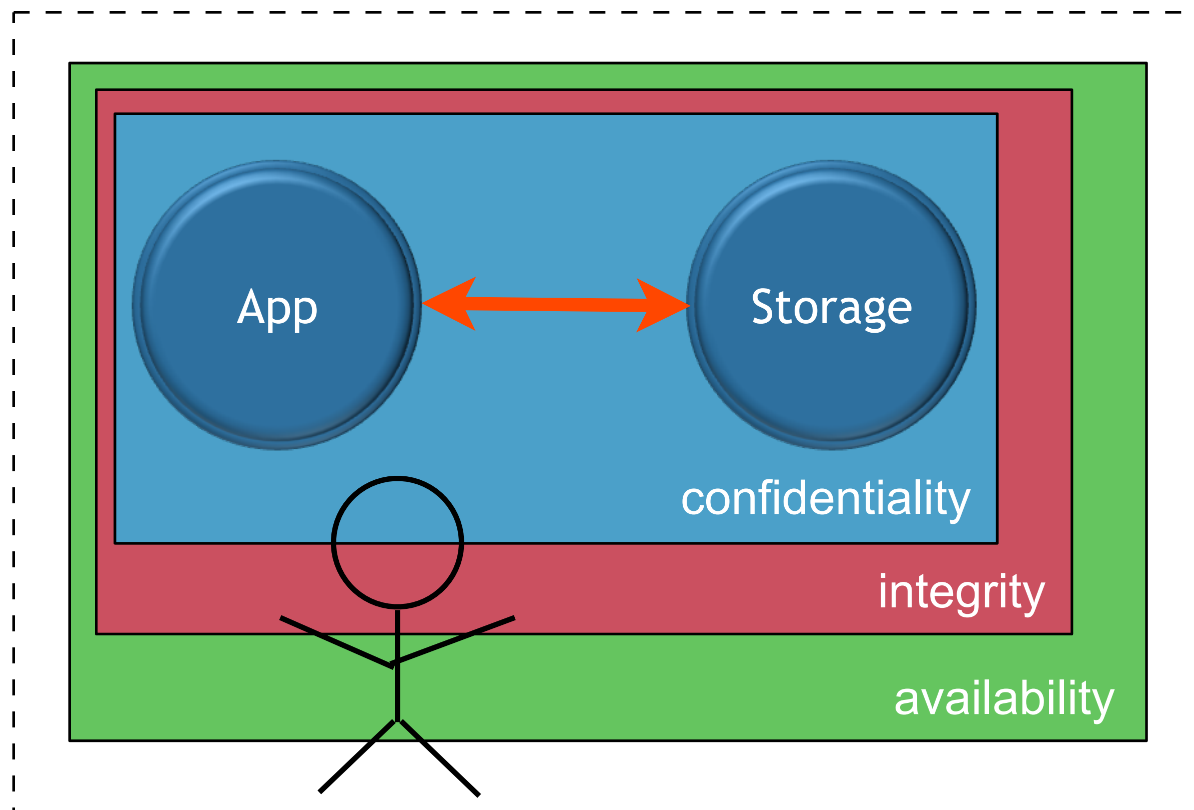
Cloud Storage is Cheap, Easy, and Scalable

- Plenty of vendors: Amazon, Rackspace, Google
- But it changes the security picture
  - who else can see your data?
  - who else can modify your data?

It is increasingly easy to outsource the storage of bulk data. Economies of scale allow storage providers to host data at lower cost than you could do internally, literally pennies per gigabyte. Scaling is easier, and intermittent capital expenses turn into predictable monthly operational costs.

But it changes the security picture.

Property Perimeters

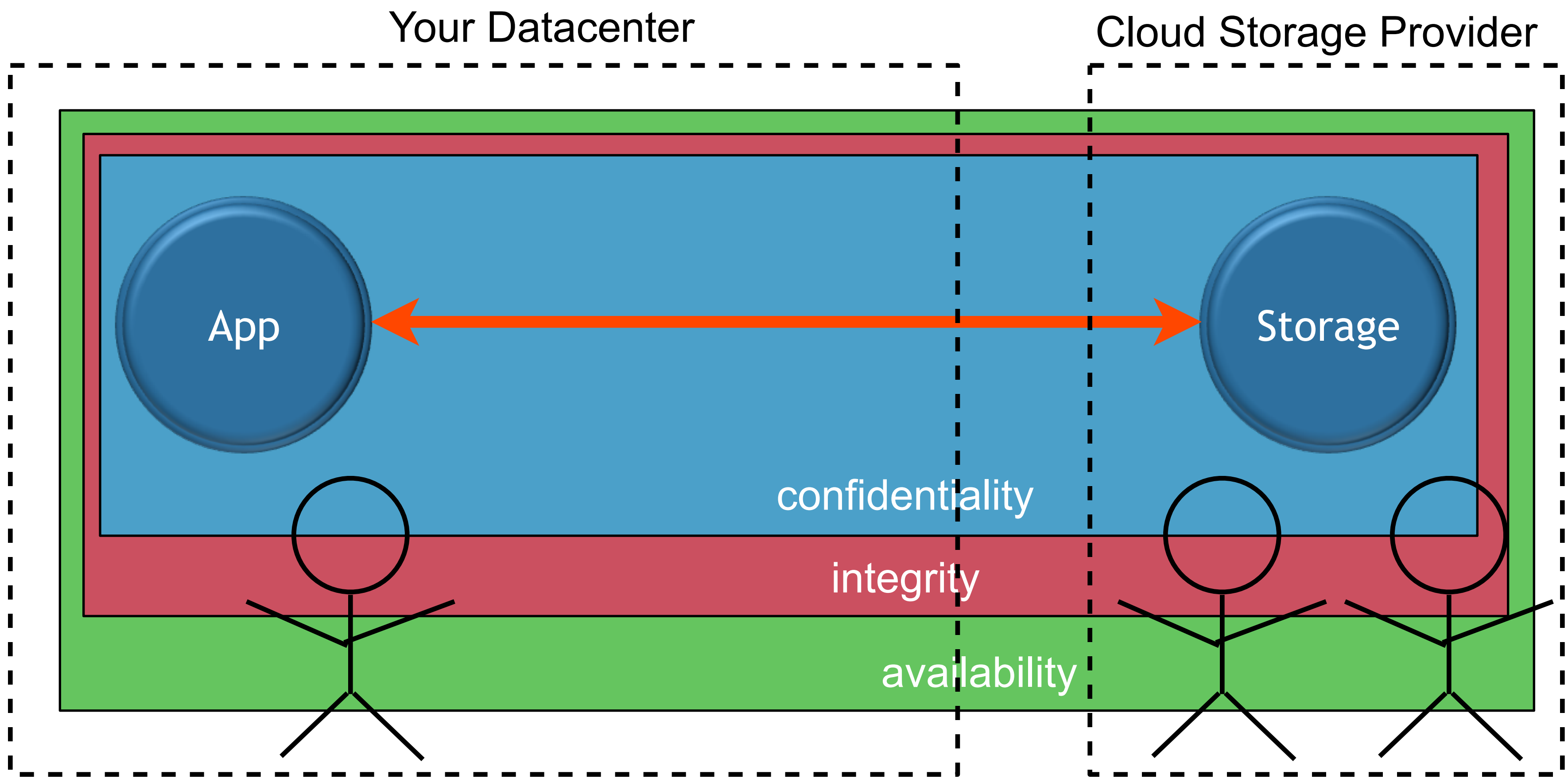


Everyone understands the notion of a security perimeter. It's like a wall around the important things: you rely upon everything inside it. Your security depends upon everything inside this wall behaving as expected.

We can refine this to talk about separate perimeters for separate properties. In the case of storage, we care about three things: confidentiality, integrity, and availability. Confidentiality is knowing that nobody else can see your data. Integrity is knowing that you get the right data, not something that's corrupted. And availability is that you get your data at all, quickly, any time you want it.

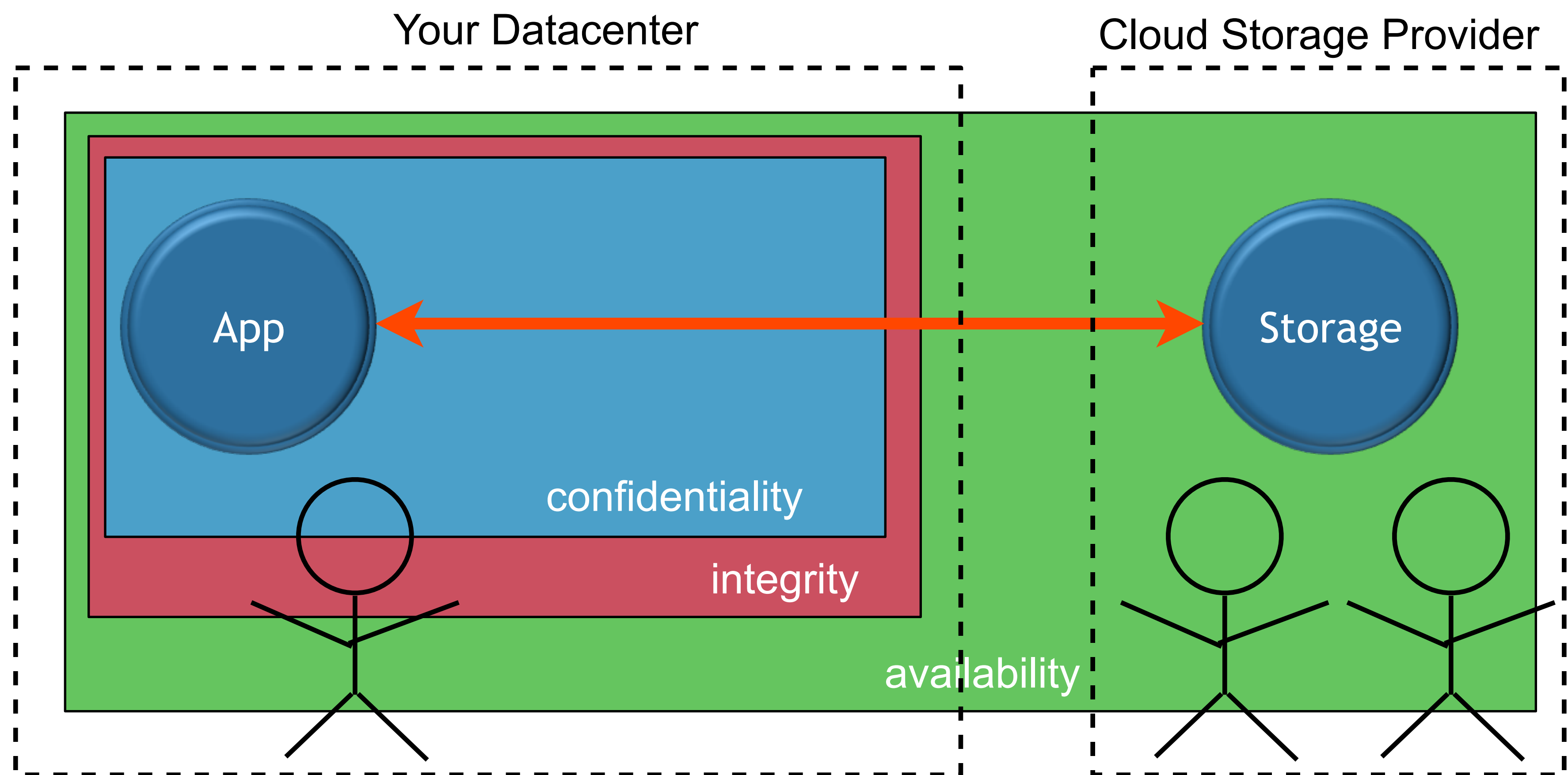
For data that you manage on your own hardware, you only get these properties if all of your own hardware works correctly and remains uncompromised. This is hard enough: you're reliant upon every admin, every employee, every janitor who can get at those machines to do the right thing all the time. You're **also** reliant upon your attackers to do the right thing by failing to break in to your machines.

Drawing Perimeters Around Clouds



When you include outsourced storage, all these perimeters are stretched. In addition to your own hardware and staff, you are now vulnerable to failures or compromises of your storage providers facilities. How many people can see your data now? Can you even count them? What sorts of assurances can you possibly have? You’re making an economic tradeoff between cost, convenience, and security, but with hardly any information about one of these important factors.

Separate the Perimeters



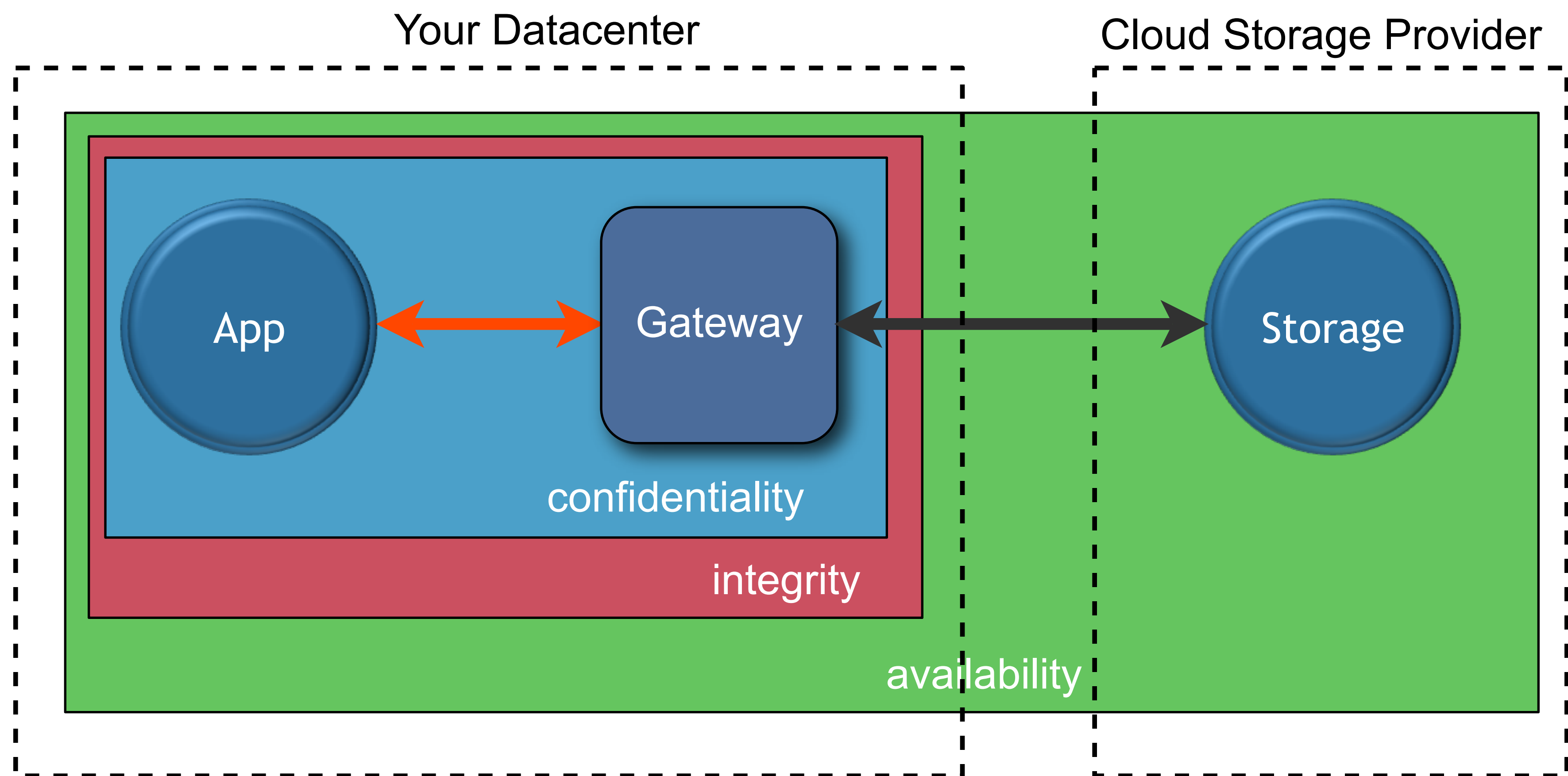
So our goal is to separate these concerns. There has been a lot of discussion about what sorts of security the cloud provider should be obligated to give you. Our position is that you should assume they give you nothing. Purchase availability from your storage provider, but bring your own security.

(metaphor about a celebrity who hires a limo, takes a commercial flight, but brings their own bodyguard instead of accepting one from the limo/airline company)

We want to build a system in which your data remains confidential even if the storage provider publishes everything you give them to the entire world. And which can detect even a single bit flip in that data.

With a system like that, you're making a cost-benefit analysis based upon the provider's ability to offer availability alone, which is something you can actually measure. This is a much easier decision to manage.

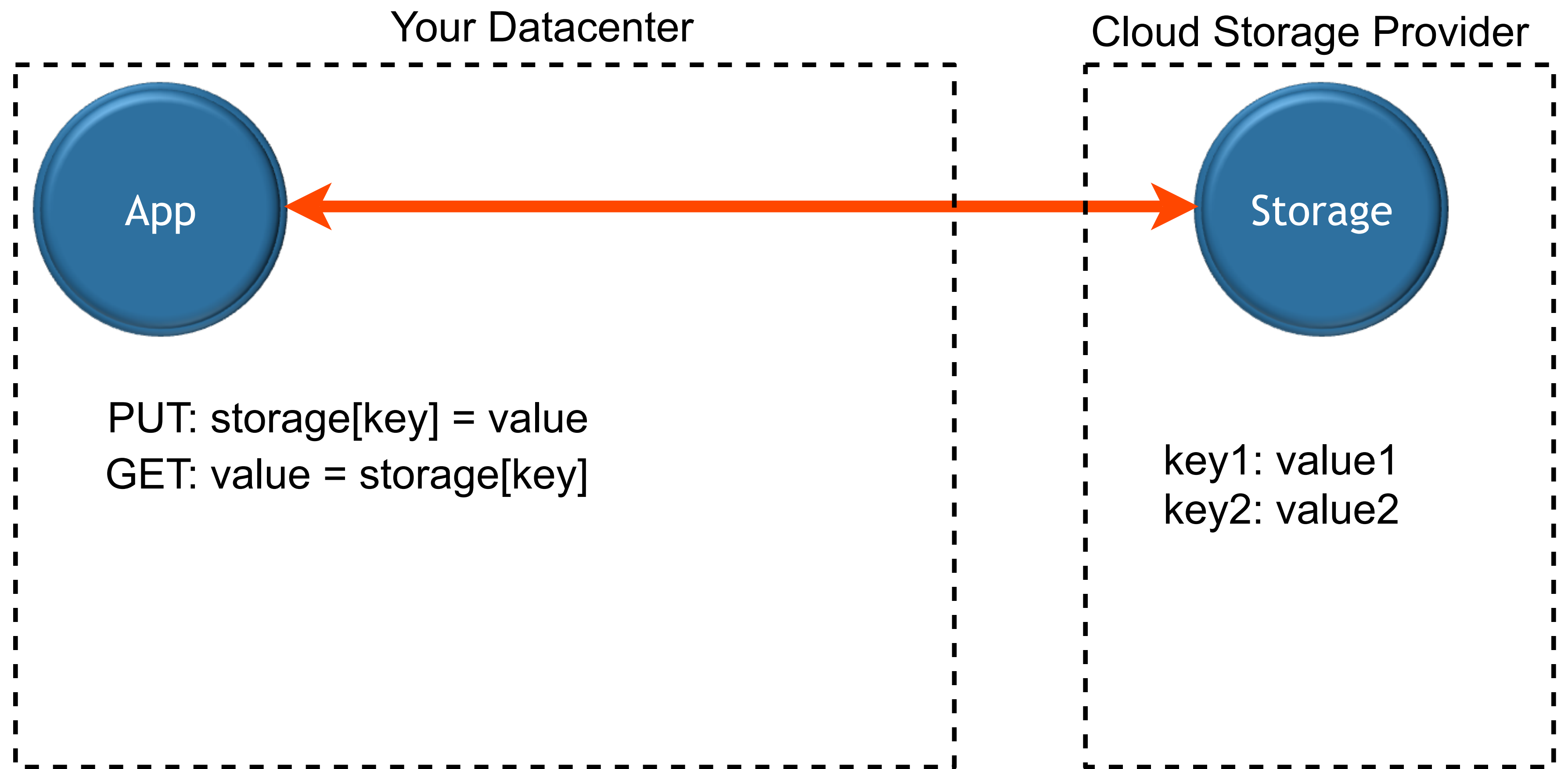
Gateway



Basically we want to implement a gateway, within your own security perimeter, that performs encryption and integrity checking between the plaintext that your app speaks (red arrow) and the validated ciphertext that you give to your storage provider (black arrow).

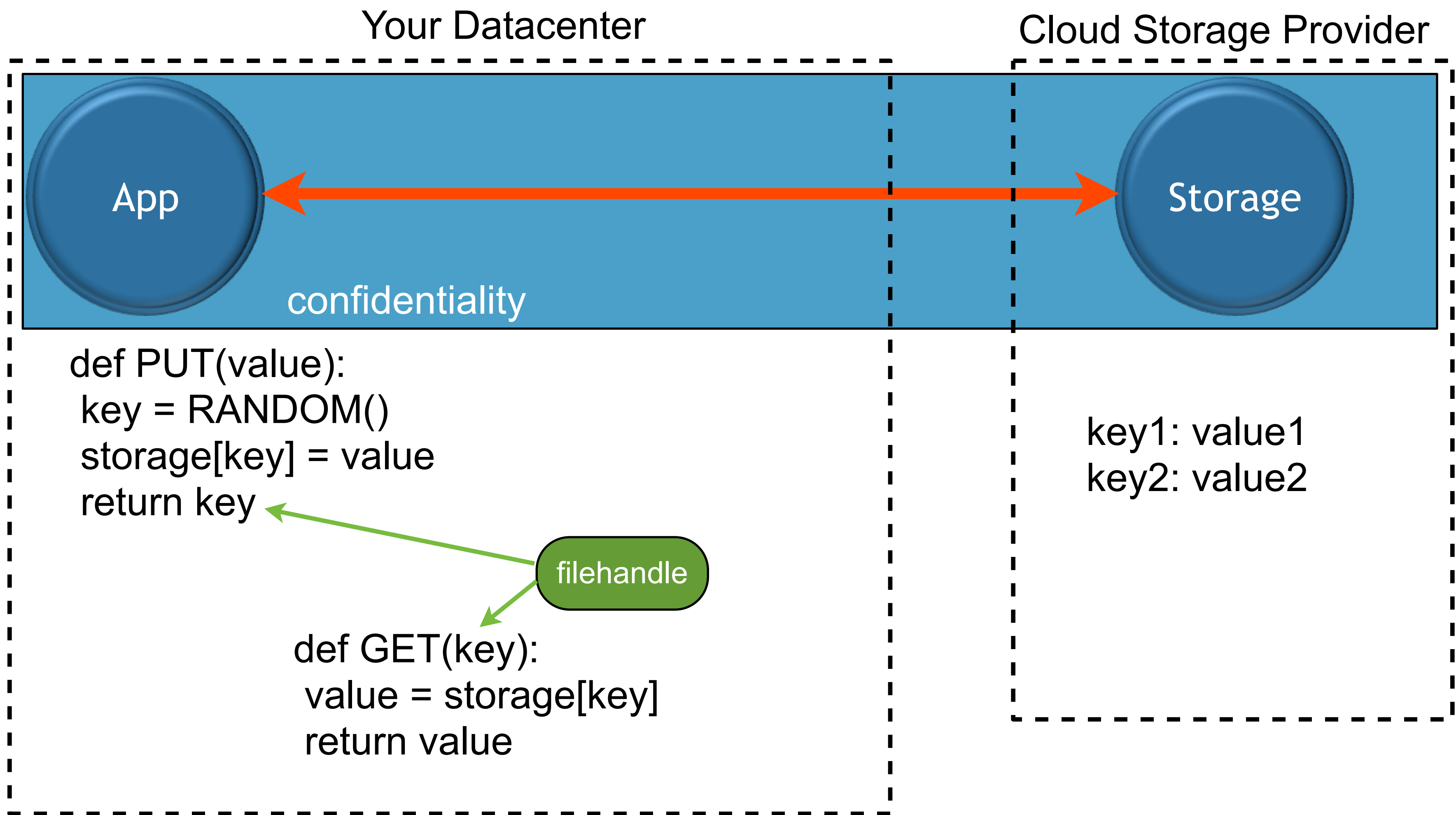
Bring Your Own Security

Key-Value Store



So let's make this a bit more concrete. A common API for storage services is the Key-Value store. There are two basic operations. You can PUT a key with a value, and you can GET the value for a previously stored key. The key can be an arbitrary string, and the value is an arbitrary blob of data.

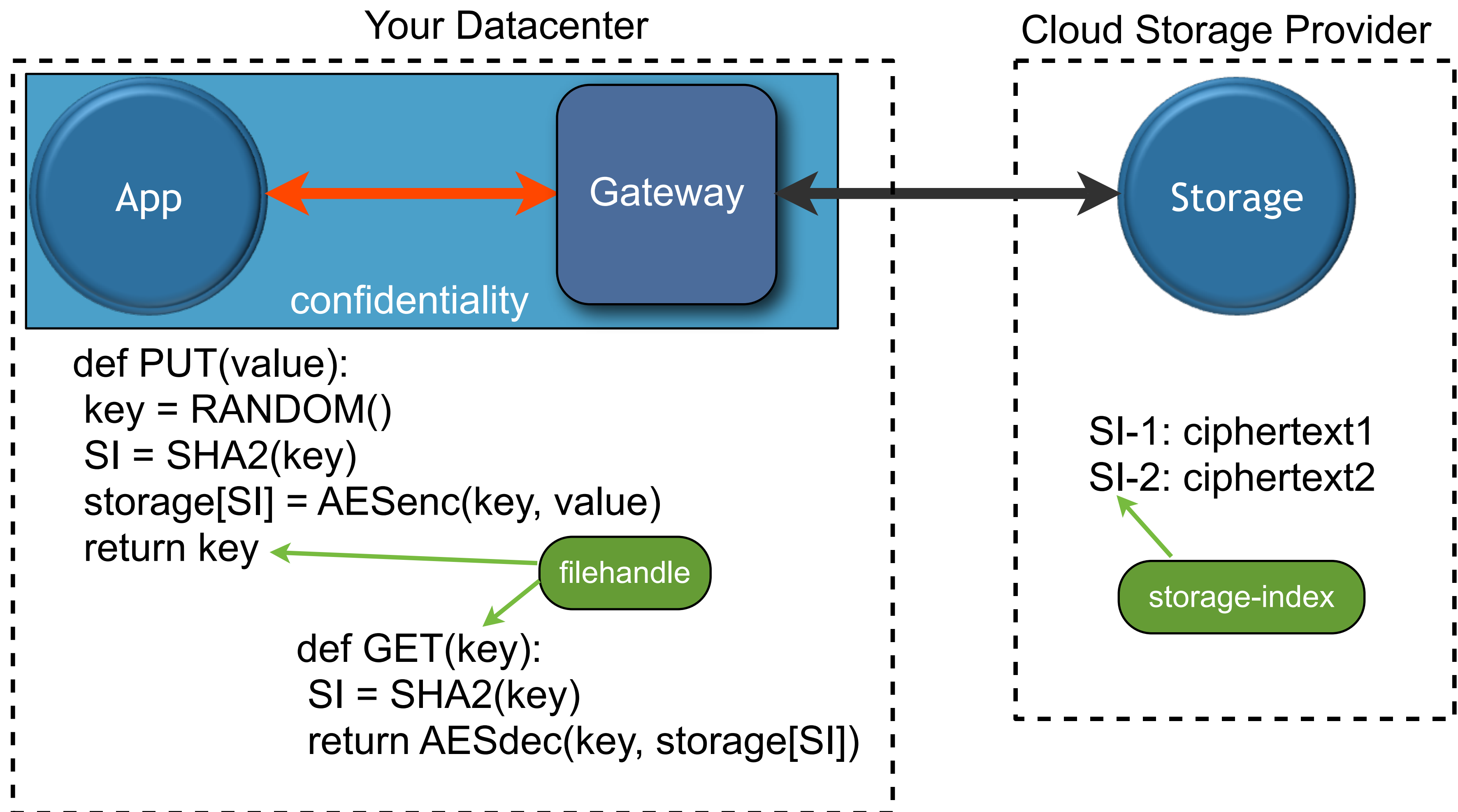
Opaque Key-Value Store



It's increasingly common for storage systems to give you back an arbitrary key instead of letting you choose your own. Many Content Distribution Networks (CDNs) use this technique. You can think of the key as an opaque file-handle for one particular piece of data. The application doesn't care what it is, it just knows to hand it back to the GET method later on. They are usually stored in some other data structure, like a database "foreign key" column.

Note how the storage system lies within the confidentiality perimeter: any compromises of the storage provider will result in a loss of security. This is how most current cloud storage systems work.

Encrypt Before Store

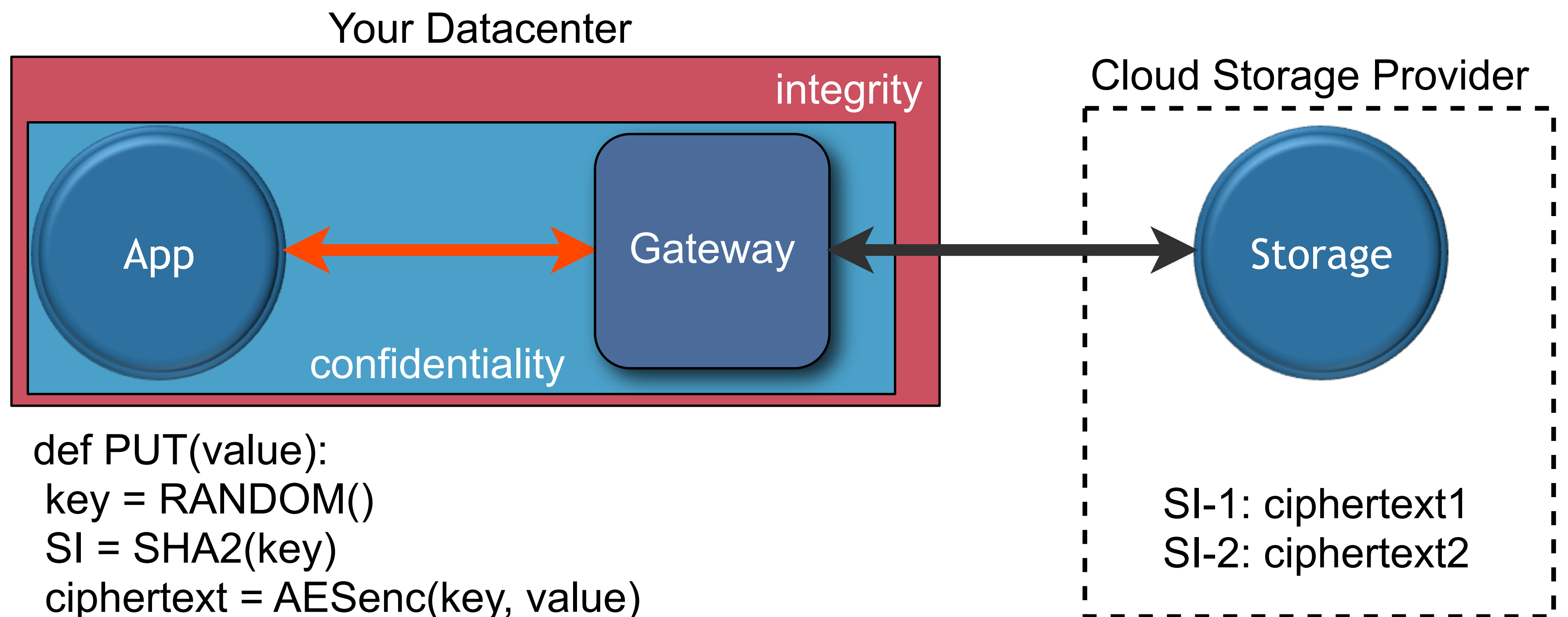


So our first step is to insert a gateway, which encrypts the data before giving it to the storage system. We generate a new unique AES key for each file, and return that key to the application as the filehandle. We derive a “storage index” from that key with a one-way hash, to tell the storage provider where to store the ciphertext. This saves us from needing to remember the storage index separately for each file.

Note that by storing the key **in the file-handle**, much of the “key management” problem goes away: if you have the filehandle, you have all the information you need to locate, retrieve, and decrypt the file. The application uses a filehandle to access the plaintext; the storage server uses a storage-index to access the ciphertext. From the application’s point of view, nothing has changed: we’re just building the filehandle in a different way than before.

This removes the storage system from the confidentiality perimeter. Nothing the storage host can do will compromise the confidentiality of our data, because they never get the decrpytion key. We are still relying upon it for integrity: a bit flip in the cloud will be decrypted and result in corrupted data arriving to our application.

Encrypt, Hash, Store




```
def PUT(value):
 key = RANDOM()
 SI = SHA2(key)
 ciphertext = AESenc(key, value)
 storage[SI] = ciphertext
 filecap = (key, SHA2(ciphertext))
 return filecap
```

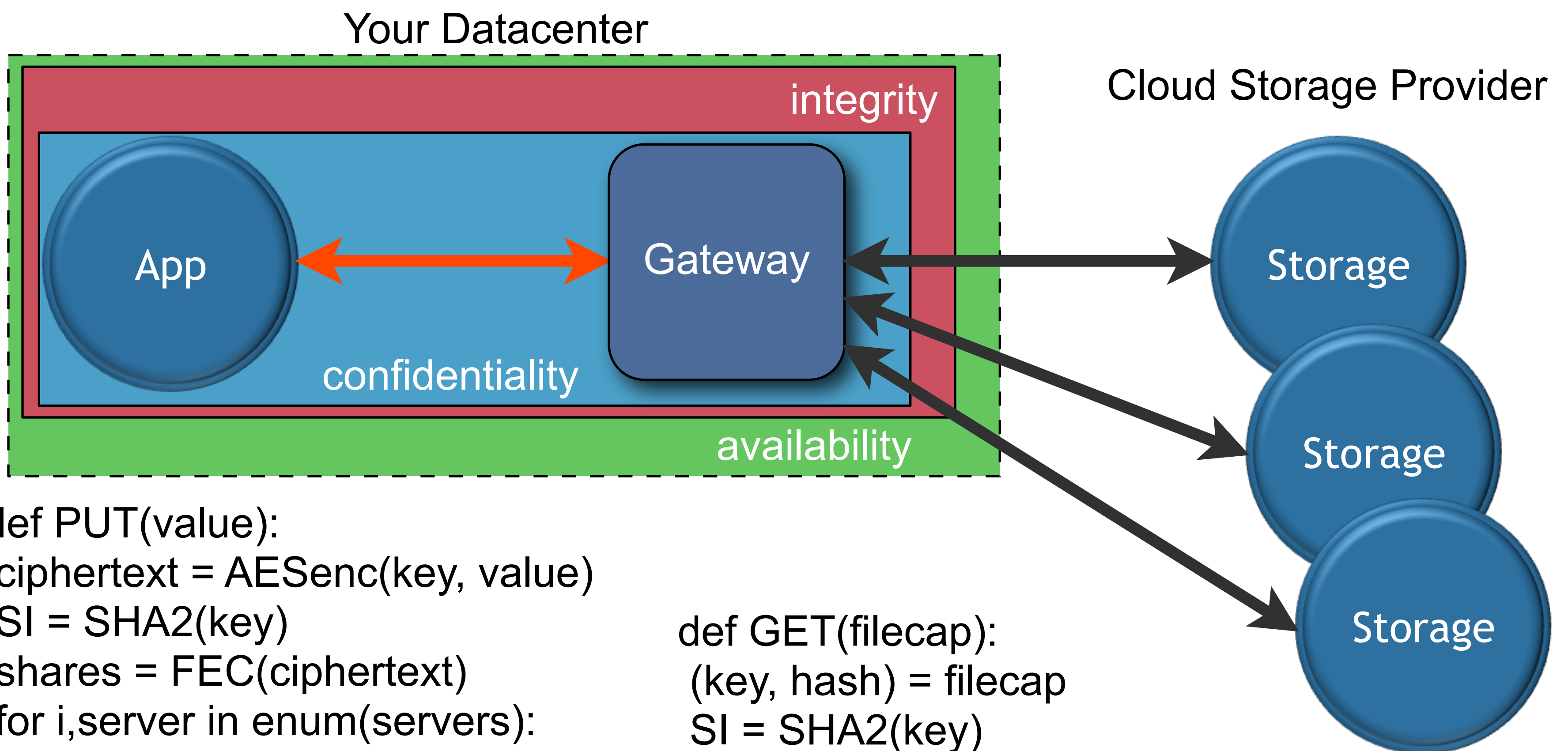
```
def GET(filecap):
 (key, hash) = filecap
 SI = SHA2(key)
 ciphertext = storage[SI]
 assert(SHA2(ciphertext) == hash)
 return AESdec(key, ciphertext)
```


We can protect our data's integrity against errors in the storage system by hashing the ciphertext before delivery, and checking that hash upon retrieval. A hash failure is treated identically to a failed read: availability is lost, but integrity is uncompromised. This protects the application against undetected errors on the storage host.

We store the hash next to the encryption key, inside the filehandle. At this point, we start calling the application-side retrieval handle a "filecap", since it provides the **capability** to retrieve the file. It is just a string, containing two cryptographic values. Note that this filecap is both necessary and sufficient to retrieve the file.

We hash the ciphertext (as opposed to the plaintext), for reasons we'll go into later.

Erasure Coding for Reliability

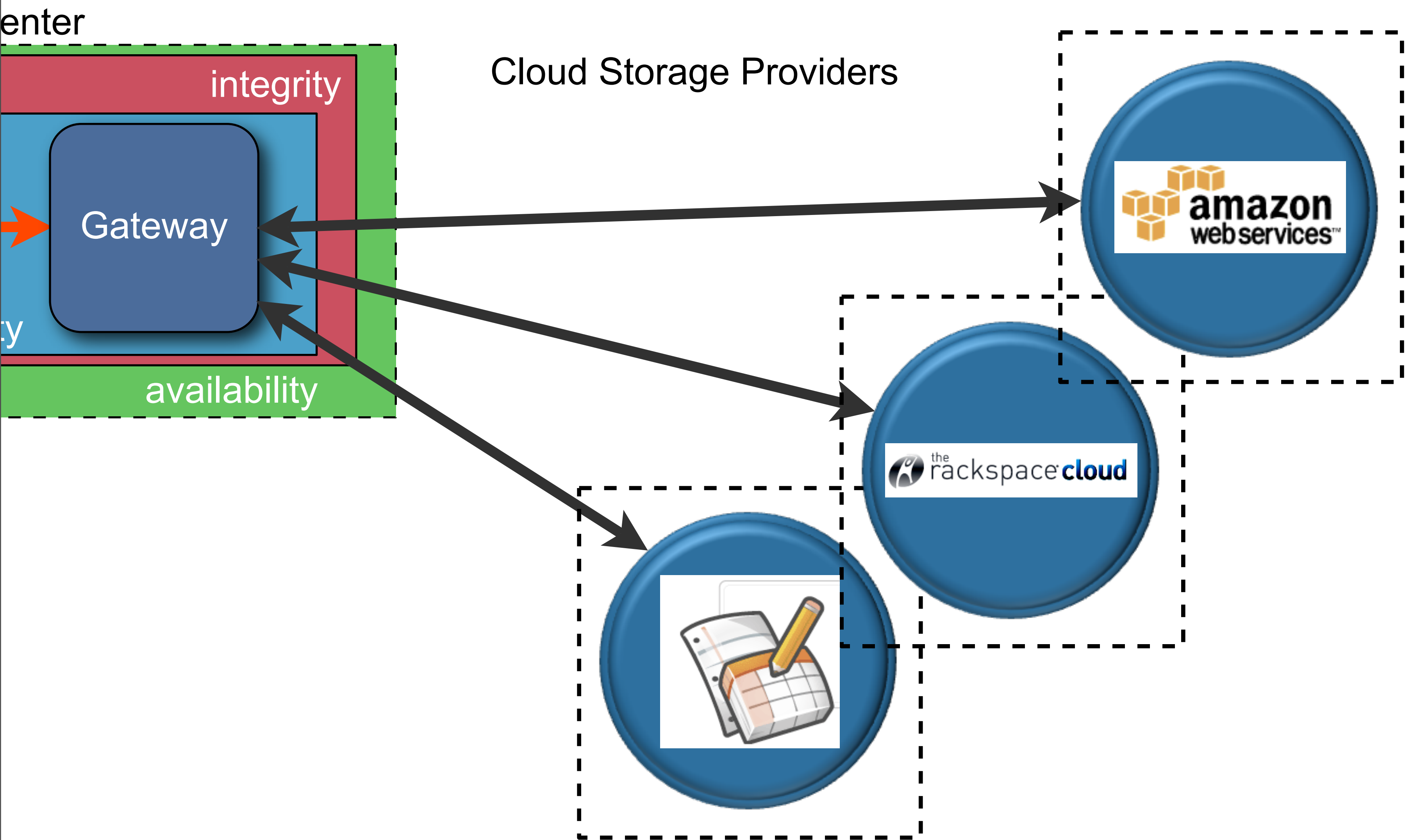



```
def PUT(value):
 ciphertext = AESenc(key, value)
 SI = SHA2(key)
 shares = FEC(ciphertext)
 for i,server in enum(servers):
   server.storage[SI] = shares[i]
 filecap = (key, SHA2(ciphertext))
 return filecap
```

```
def GET(filecap):
 (key, hash) = filecap
 SI = SHA2(key)
 shares = someservers.storage[SI]
 ciphertext = unFEC(shares)
 assert(SHA2(ciphertext) == hash)
 return AESdec(key, ciphertext)
```


and for extra credit, we can apply erasure coding, also known as Forward Error Correction, to split the ciphertext into pieces, in such a way that we only need a subset of those pieces to recover the original. The Reed-Solomon algorithm is a great implementation of this, fast and simple.

We can send each piece to a different server, and thus tolerate failures of a configurable subset of them. This reduces our availability perimeter: we are less dependent upon the availability of any individual server. This might let you meet your availability goals with cheaper commodity servers that don’t individually give you super uptime. Or it might let you achieve a higher availability goal than any one server can offer. The tradeoff between cost and quality is decided by your gateway, when the file is encoded.



This technique lets you spread the risk among multiple providers. In this case, the independence of having multiple administration domains actually **helps** rather than hurts, because it gets you decorrelated failures. Instead of being vulnerable to security failures at all of your providers, your data remains available as long as at least one provider is still running. You get the maximum of their availabilities instead of the minimum of their security. This can amplify the overall reliability by a huge factor.

You’ve heard of RAID. We call this RAIC: Redundant Array of Inexpensive Clouds.

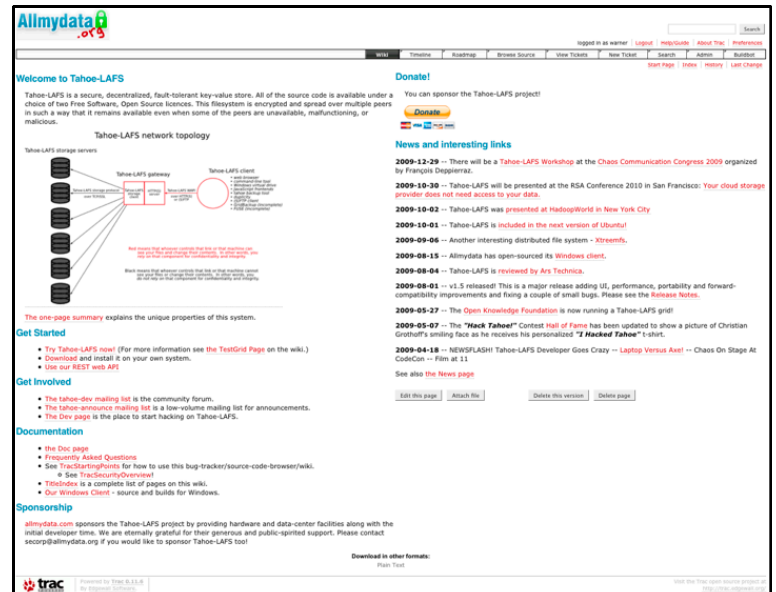
Tahoe-LAFS

zooko takes over while audience is laughing

Now that we've convinced you that you want these properties, and shown you how to build a system that provides them, it's time to show you the system that we've already built.

Tahoe-LAFS

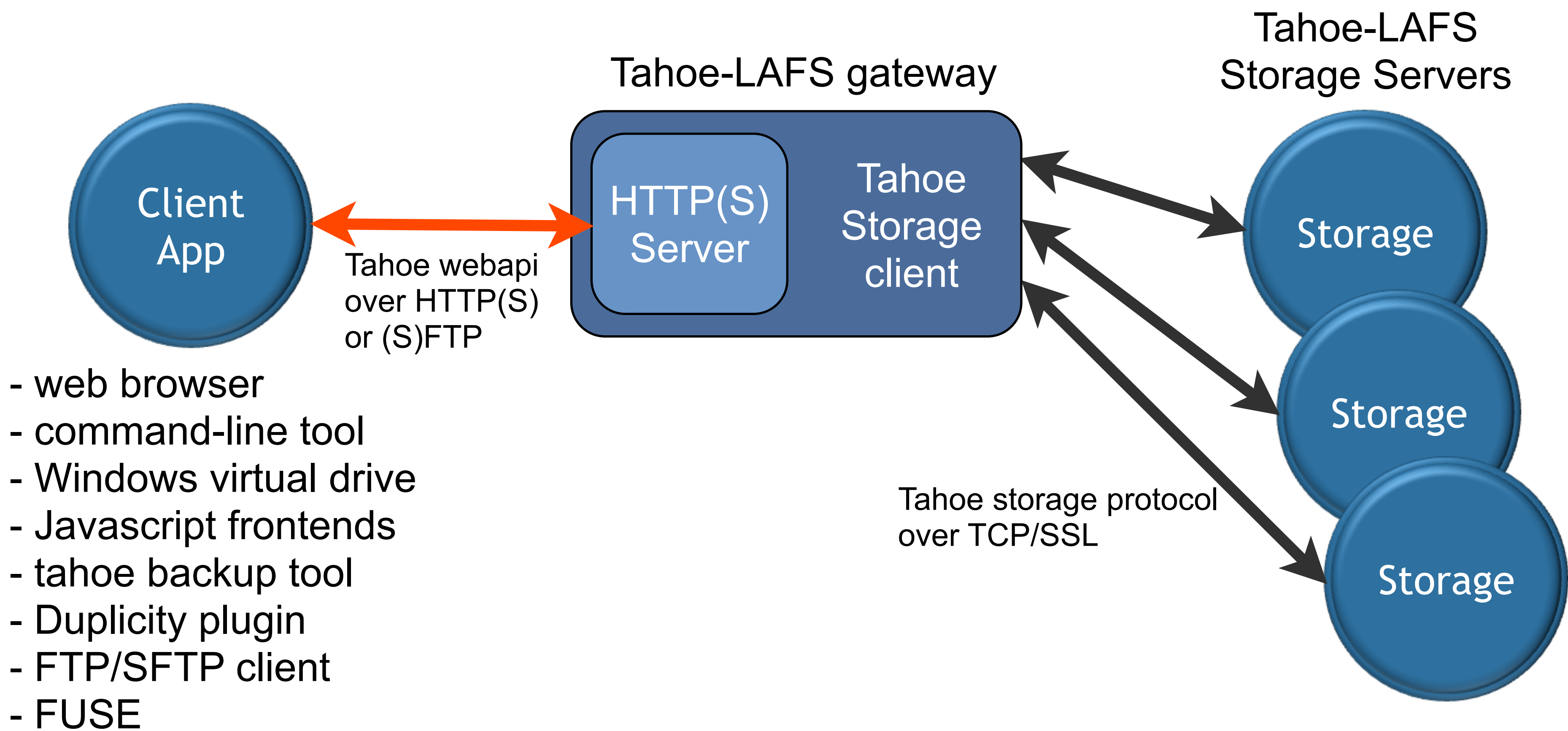
- Tahoe-LAFS: the Least-Authority File System
- http://tahoe-lafs.org
- implements distributed confidentiality, integrity, and availability
- open-source project started in 2006, as backend for a startup company offering consumer backup services



We developed Tahoe to provide backend storage services for an online backup company, in which a selling point was that the company would be unable to see its users' data.

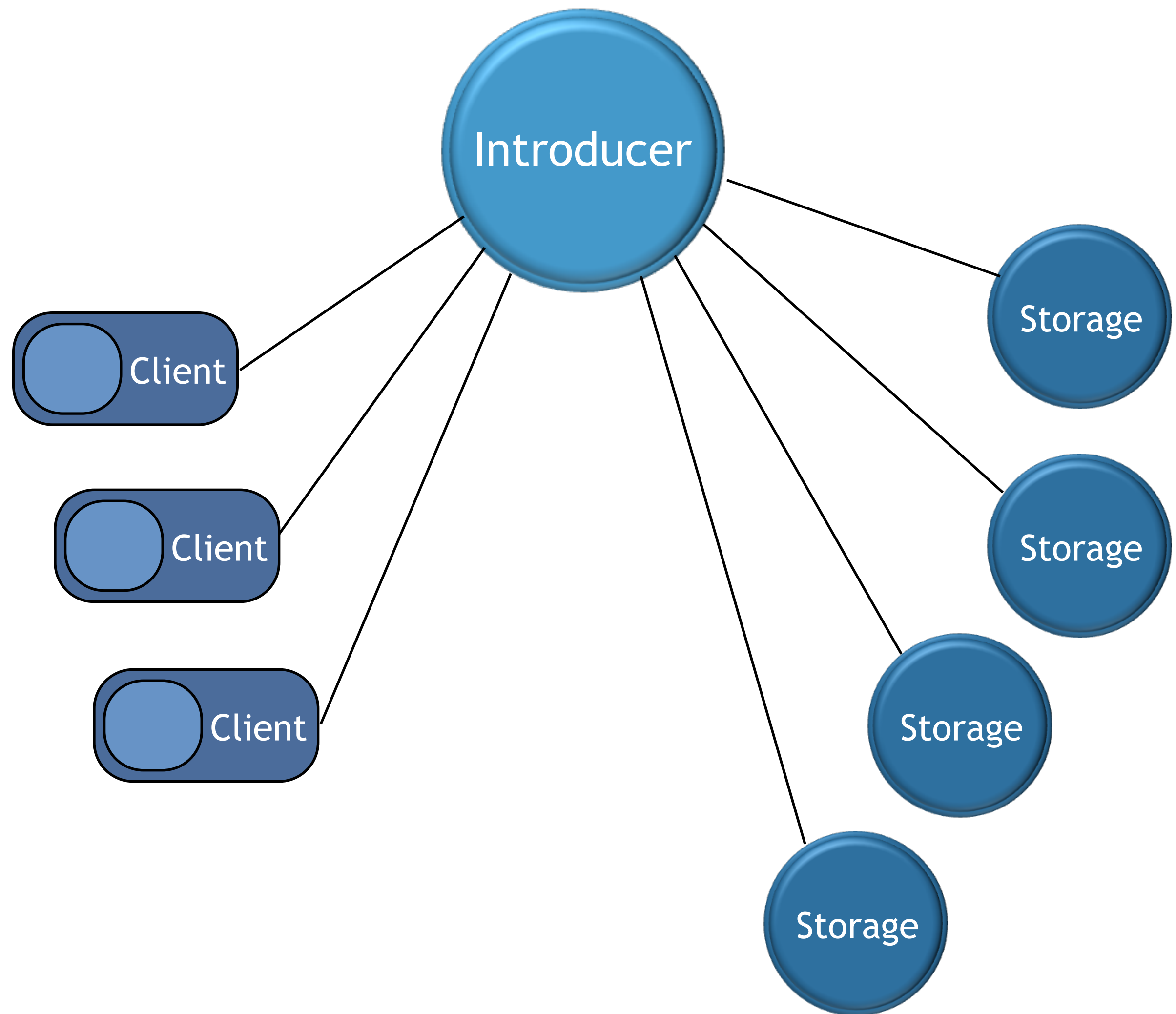
source code, installation instructions, bug tracker, mailing list, IRC channel

Tahoe-LAFS: Overview



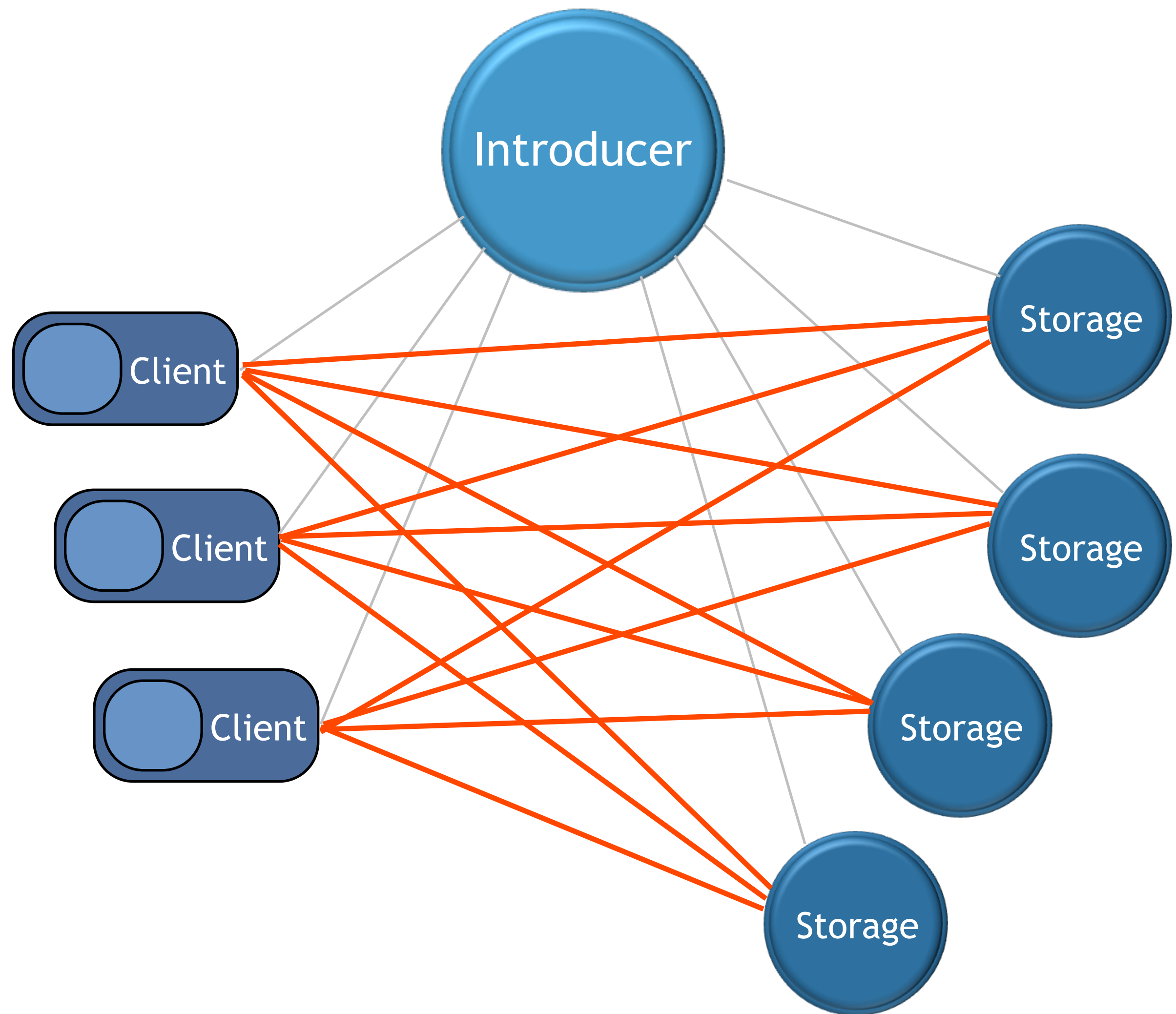
Tahoe provides both the security-providing gateway and the reliability-providing distributed storage system. The gateway contains an embedded webserver through which clients can manipulate the filesystem through a simple REST-ful HTTP protocol. This enables the creation of numerous client applications, many of which ship with the Tahoe source code.

Tahoe-LAFS Grid



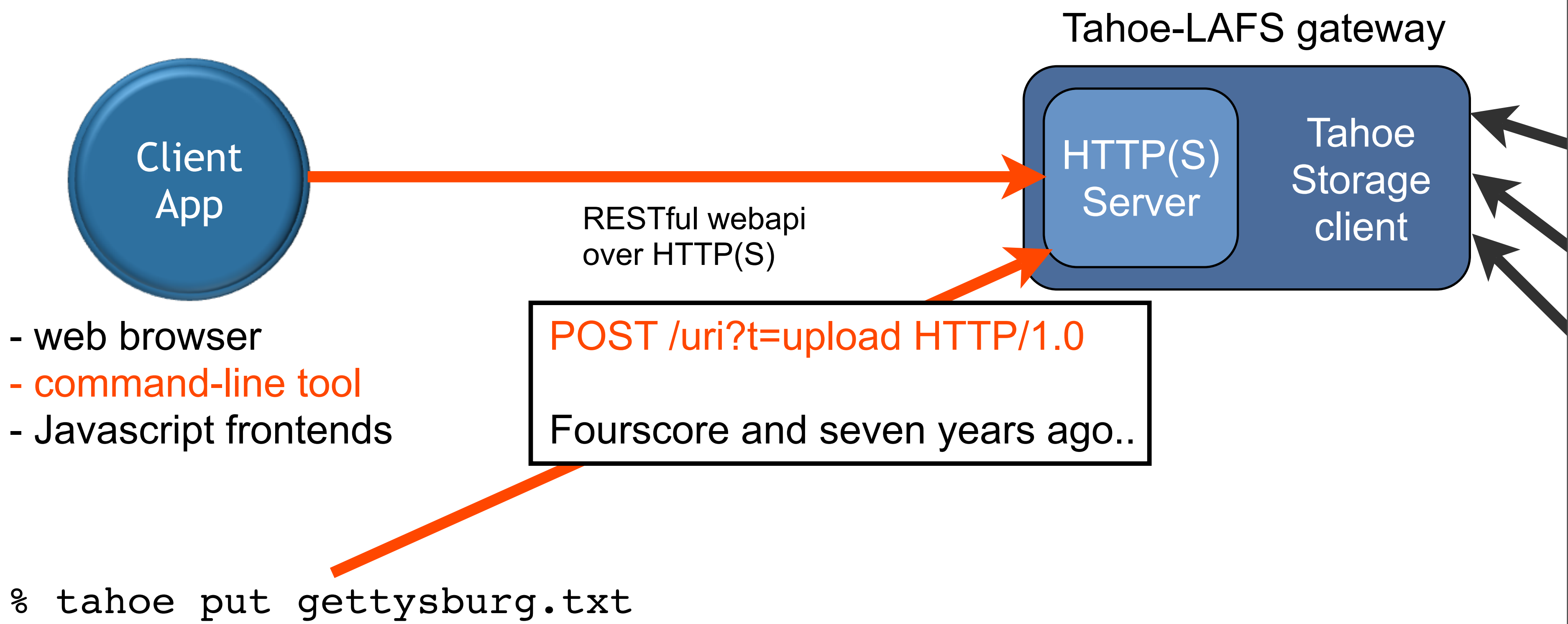
Each instance of a Tahoe grid is established by means of an “Introducer”. This is a special service that helps nodes connect to each other. All nodes connect to the introducer, both clients and storage servers. The Introducer distributes location information about all other nodes, allowing..

Tahoe-LAFS Grid



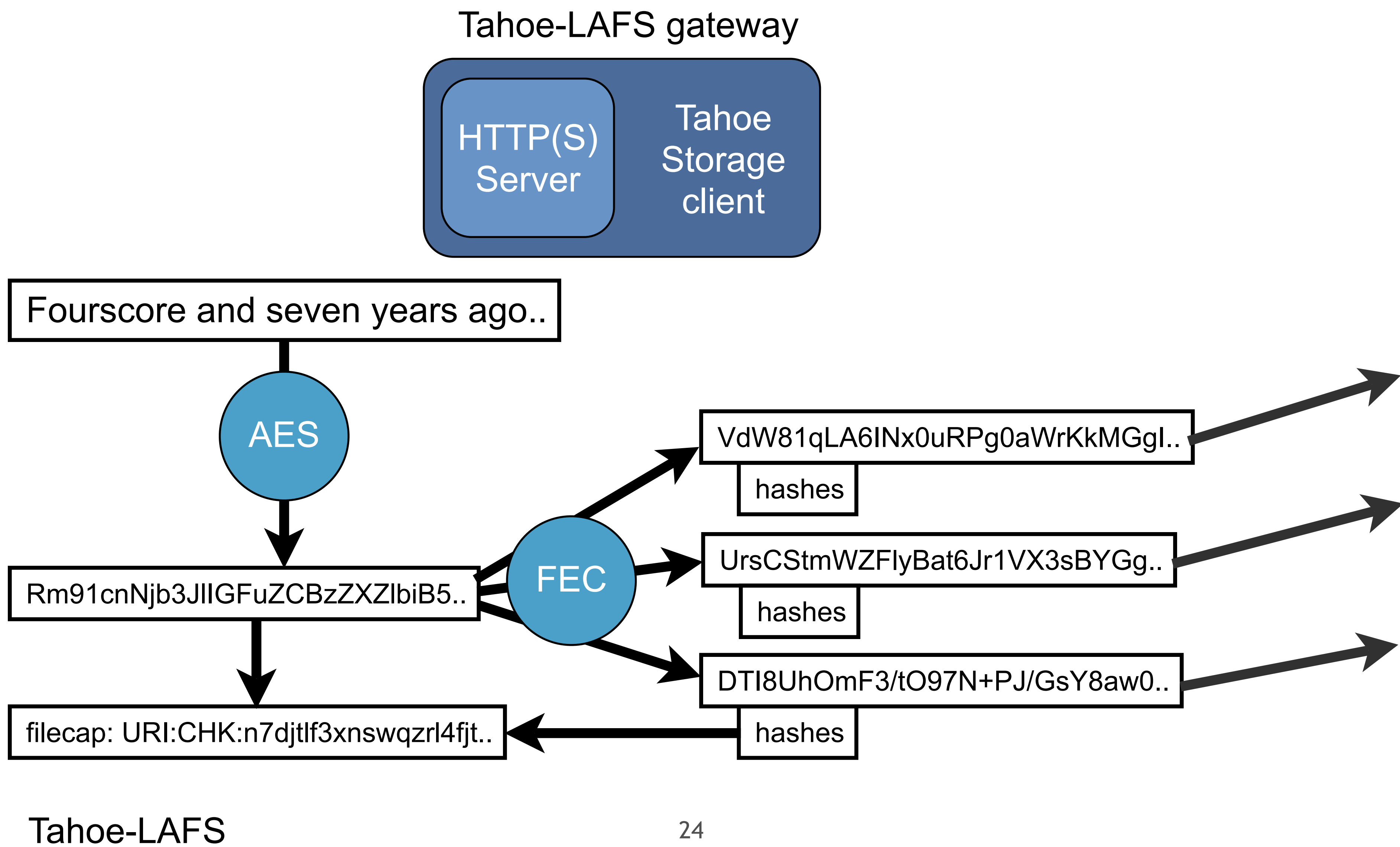
.. the establishment of a full mesh of connectivity: each client connects to all storage servers.

Tahoe CLI, webapi



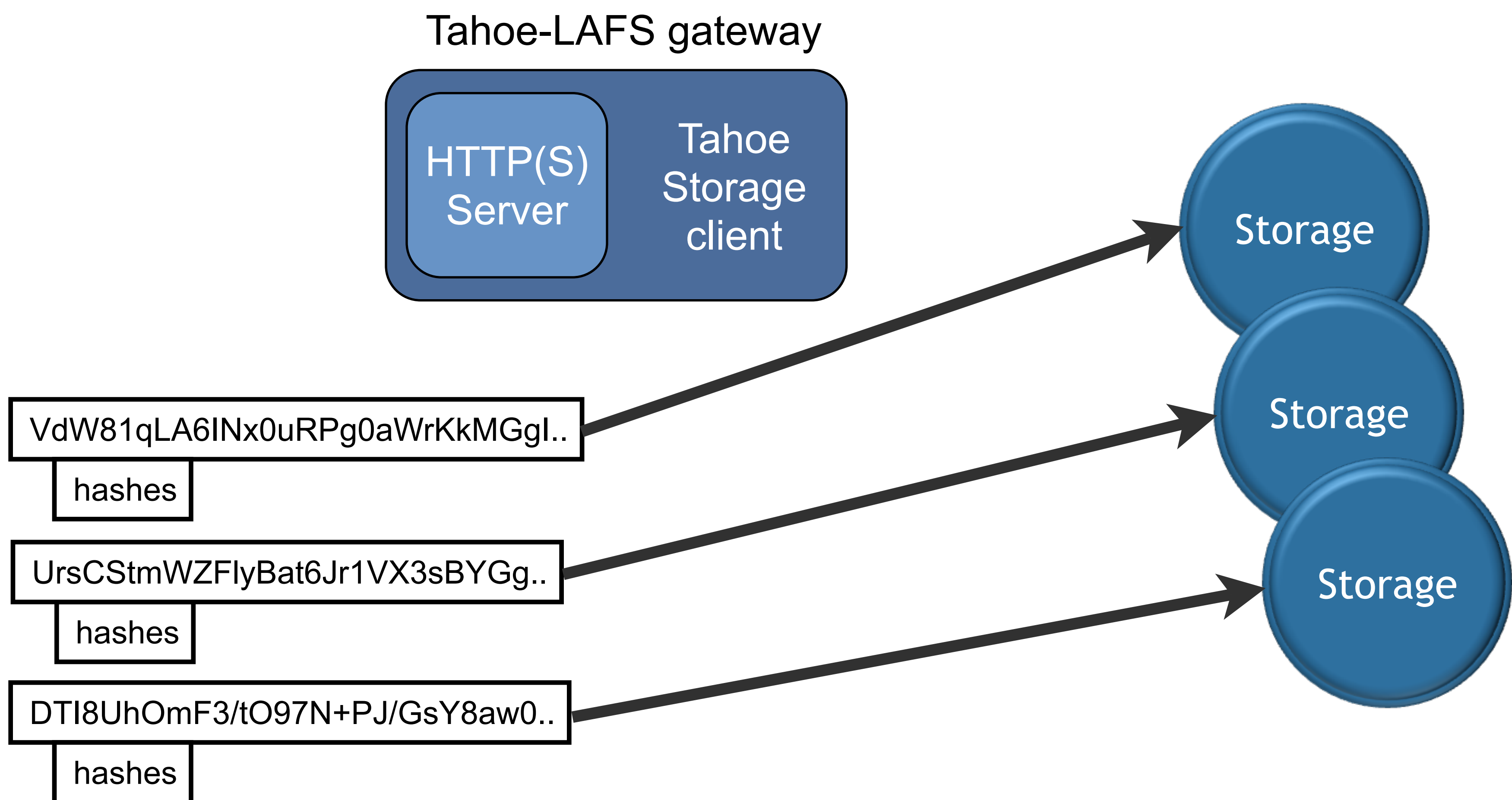

```
% tahoe put gettysburg.txt
```


The most basic operation is to upload a file into the grid through the CLI "put" command. This simply takes the input data and sends it in the body of an HTTP "POST" to a special URL hosted in the gateway process.



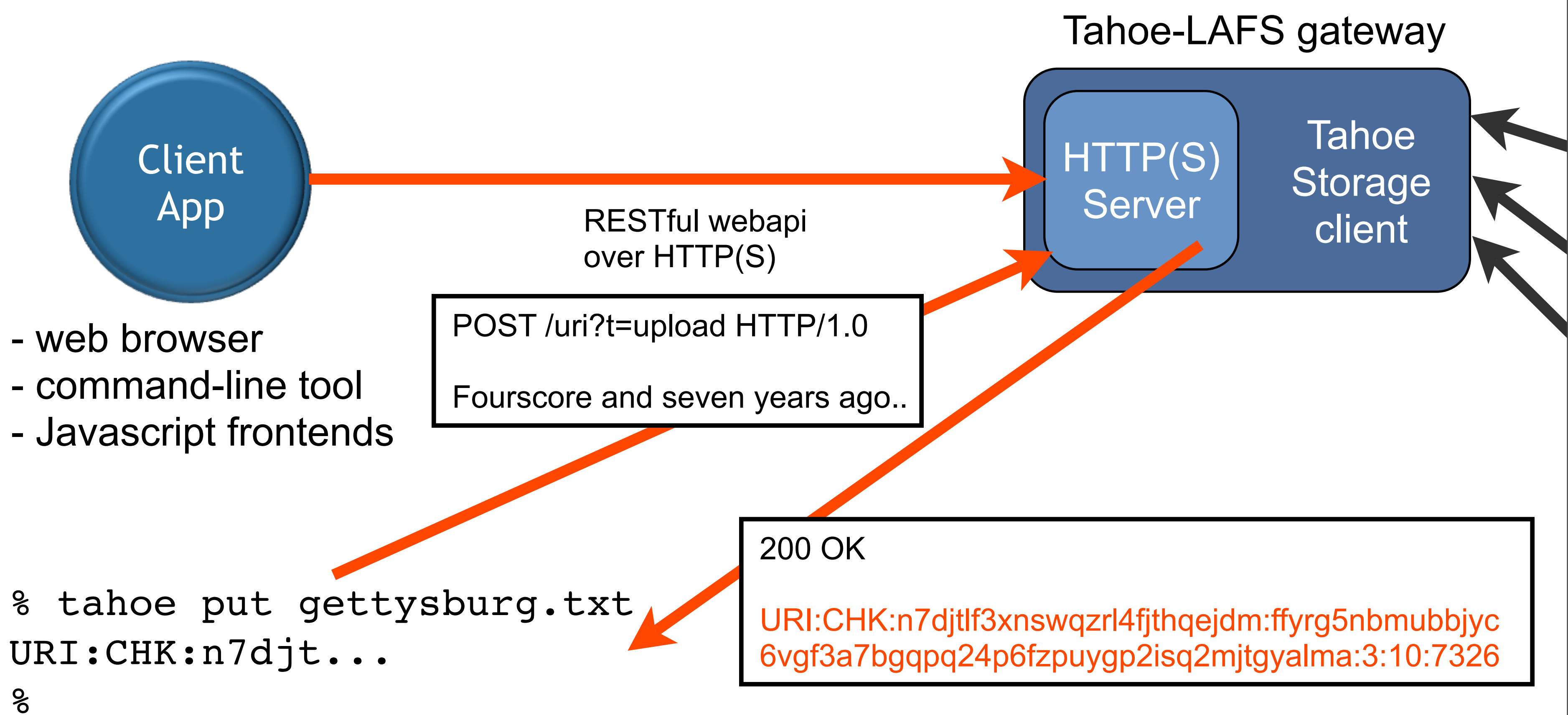
From there, the gateway performs the same steps we described earlier: encryption, hashing, erasure coding. It winds up with a collection of “shares” (of which any sufficiently-large subset will allow recovery), and the filecap. The default encoding parameters are “3-out-of-10”, meaning it creates 10 shares, of which any 3 are enough to recreate the file. Encoding can be tuned to create anywhere from 1 to 256 shares, allowing complete flexibility of the tradeoff between size expansion and reliability.

Share Upload



The shares are each sent to a different storage server, using a Tahoe-specific protocol.

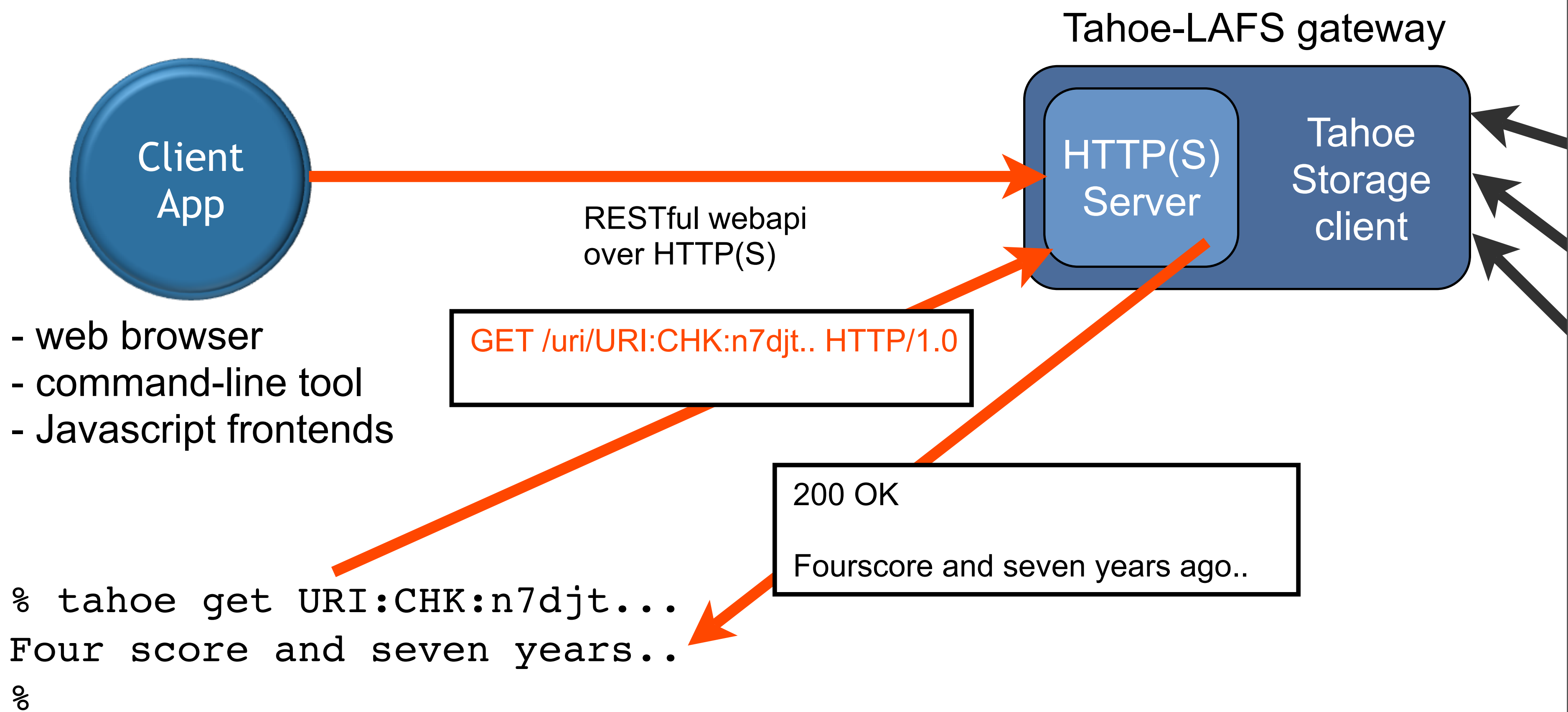
CLI returns filecap



Once the shares are placed, the gateway returns the new filecap in the HTTP response body, and the CLI tool emits the filecap.

Tahoe filecaps are fairly short ASCII strings: that's an example of a real filecap in red. These strings are easy to pass around, via IM, email, or other channels.

Downloading Files



When the filecap is passed back to the CLI "get" command, it uses the same interface to send a download request to the gateway. This finds the shares, decodes the ciphertext, verifies the hashes, decrypts, and returns the plaintext as the HTTP response body.

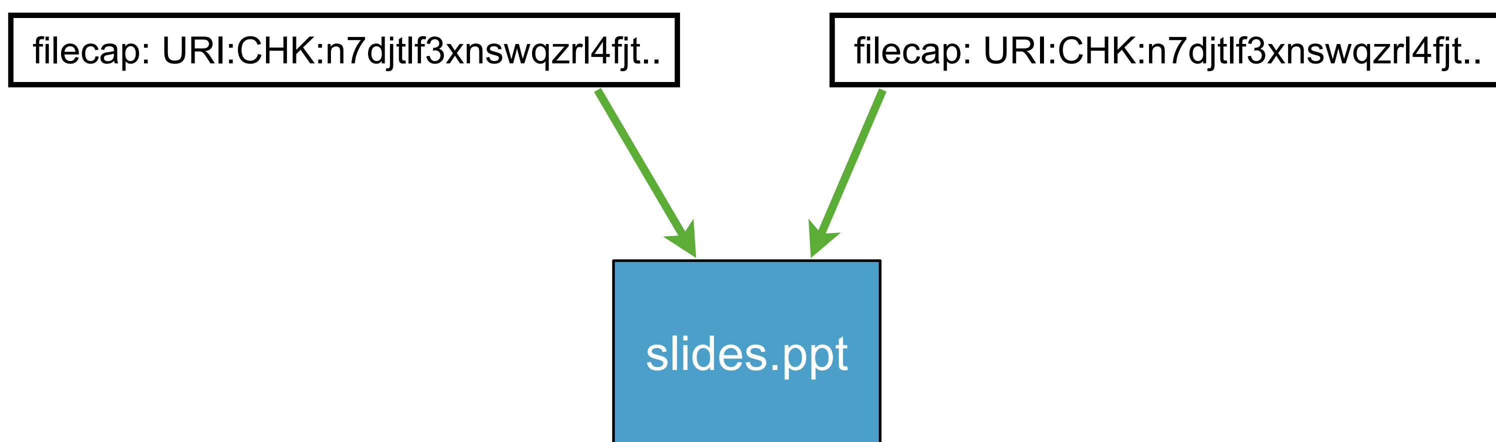
Granting access to a specific file is as easy as sharing the filecap.

Demo

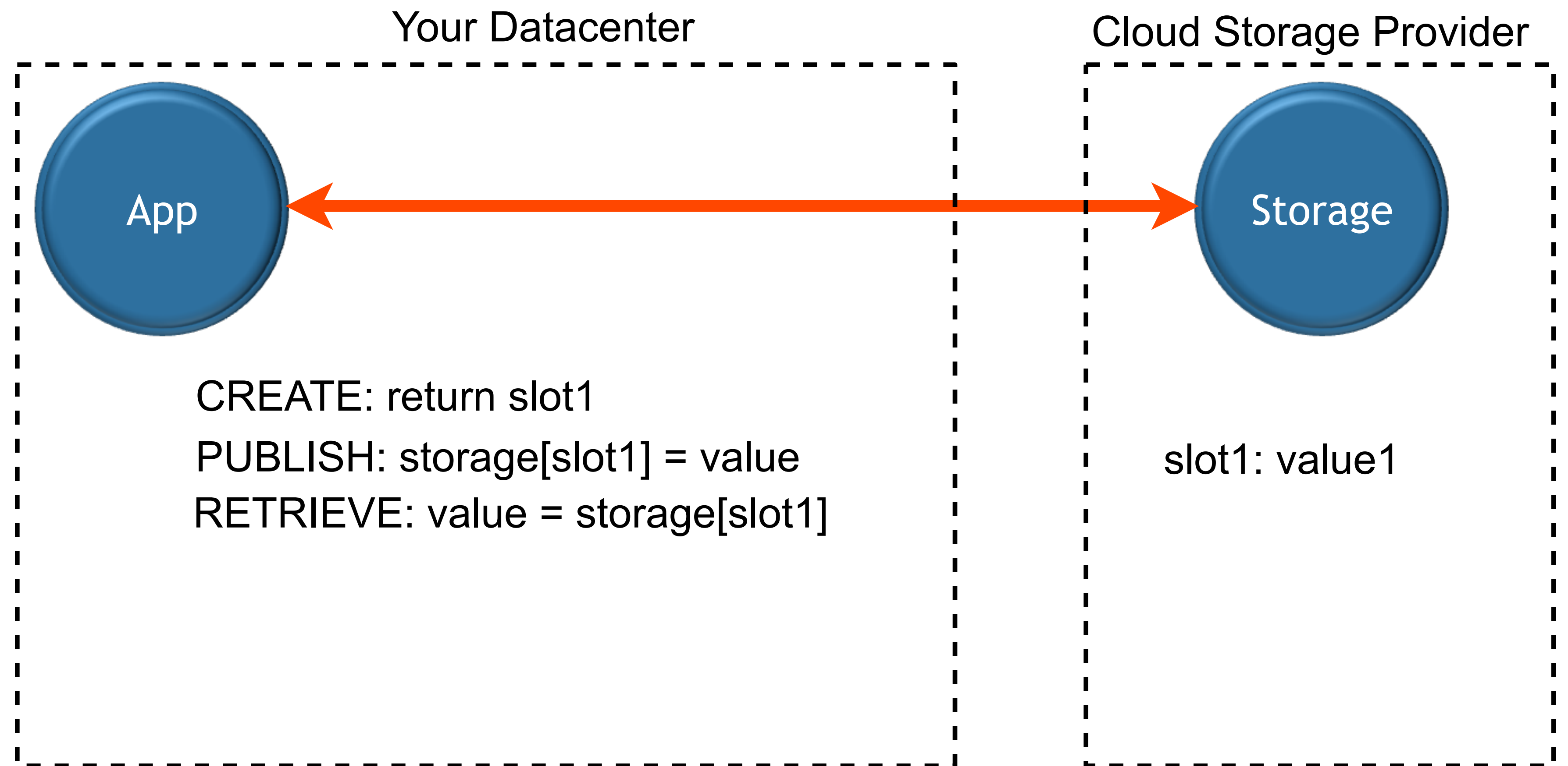
- CLI: `tahoe put`, `tahoe get`
- web interface: checker, verifier
- corrupt a share, recover file from other shares
- delete a share, repairer re-creates it

Sharing Files

- Sharing access to a single file is as easy as sharing the filecap
  - grants access to exactly that one file, no others
  - Principle Of Least Authority



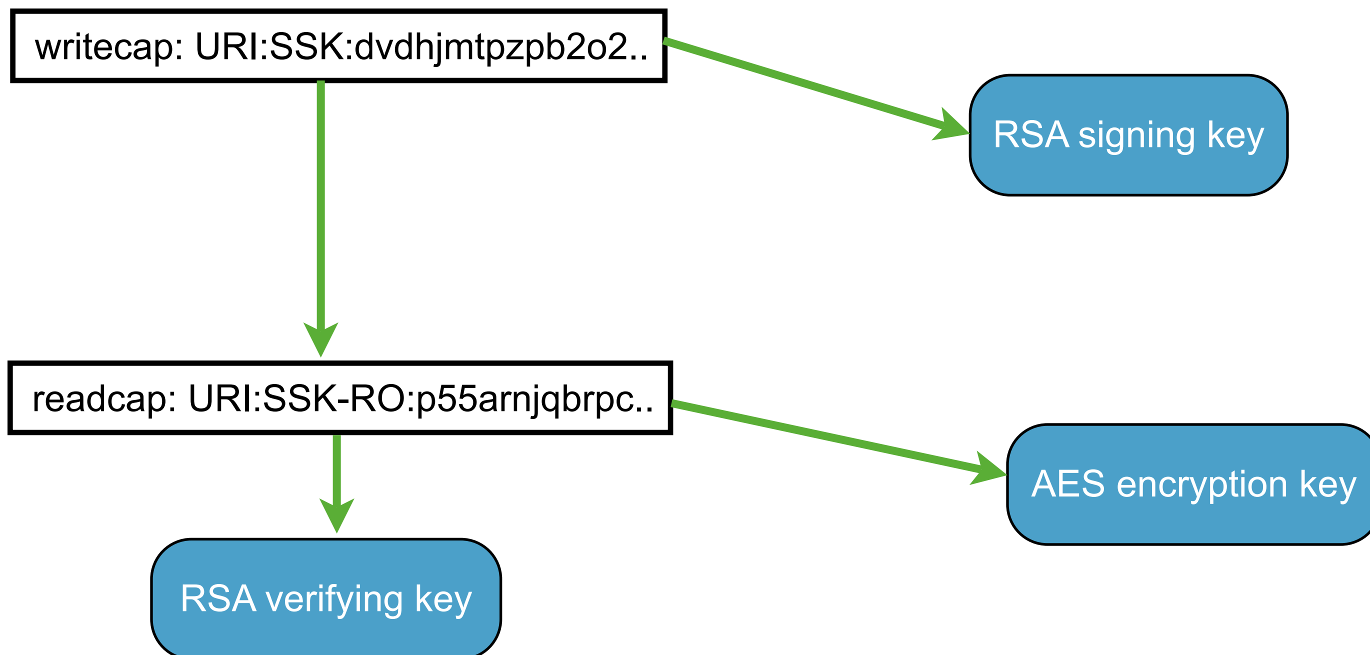
Mutable Files



Everything we've discussed until now has been about **immutable** files. These are remarkably powerful, and we've gotten a lot of milage out of them alone. But being able to change the contents of a file without changing its identity is a necessary primitive. So we also define **mutable slots**. There is an explicit **create** step, which returns the slot's identity (another opaque string). Then there are **publish** and **retrieve** operations to set and get the contents. Most cloud storage providers offer this sort of interface.

Mutable Filecaps

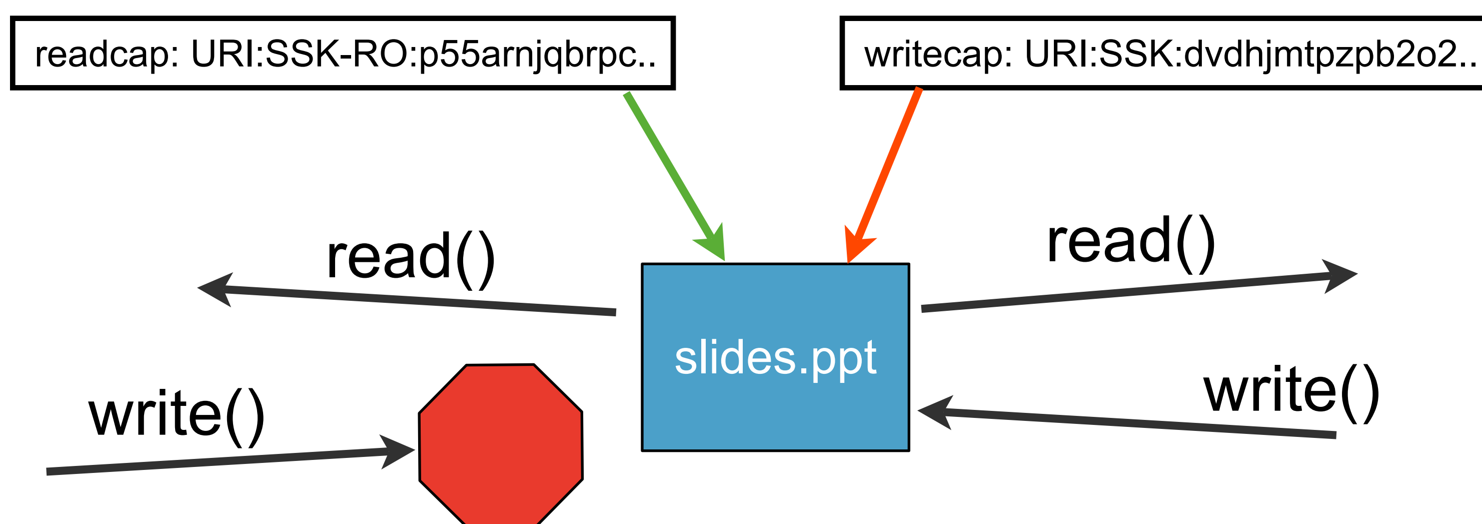
- We define two kinds of filehandles for mutable files
  - "writecaps" allow publishing new content
  - "readcaps" allow retrieving existing content
  - readcap can be derived from writecap, but not vice versa



Tahoe makes a point of distinguishing between read-only access and read-write access, and defines two separate kinds of filecaps for mutable files: **readcaps** and **writecaps**. The implementation details are a bit complex for this presentation, but basically each slot is defined by a RSA keypair. The writecap gives you access to the private signing key, which is used to sign all shares. The readcap gives you access to both the public verifying key (so you can distinguish real shares from fakes), and a symmetric AES decryption key (so you can decrypt the ciphertext into plaintext). The readcap can be derived from the writecap, giving the writer access to everything, but withholding the signing key from the reader.

Read-Only Access to Mutable Files

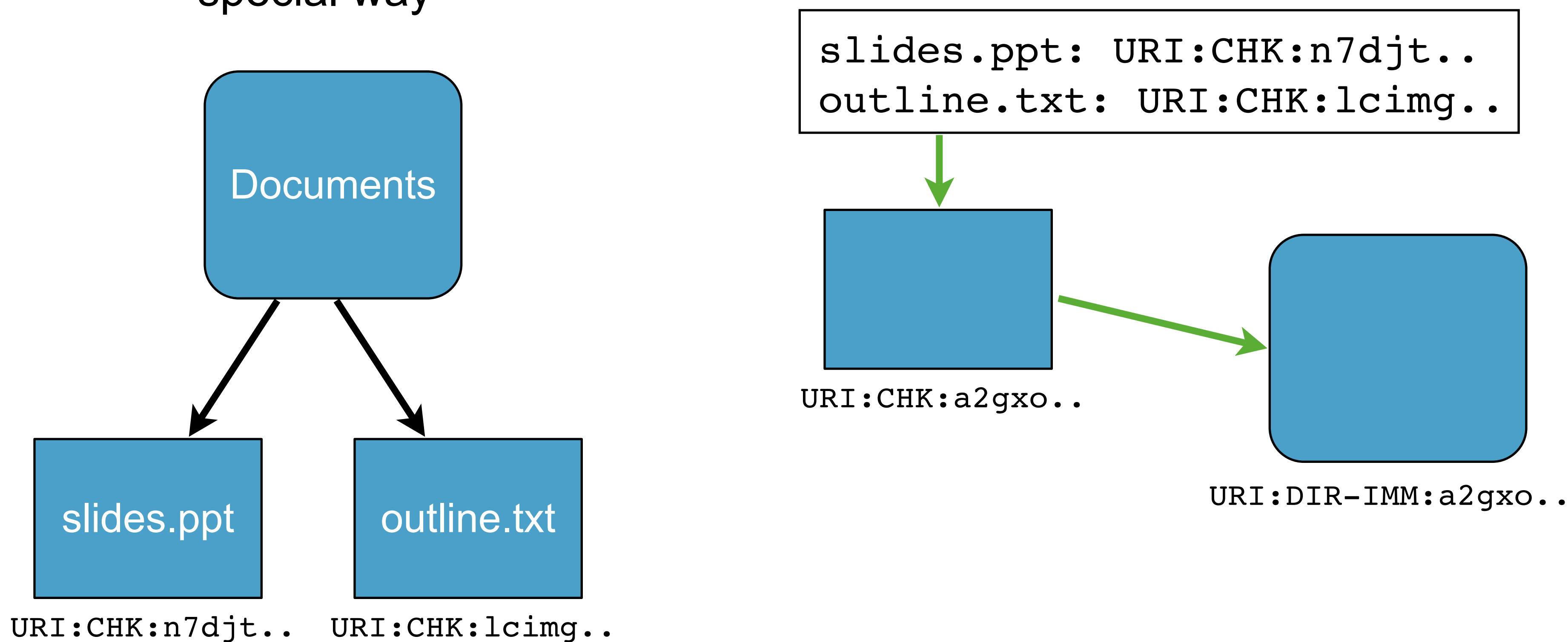
- To grant read-only access to a file, share the readcap
  - retain write access for yourself, or share it with someone else
  - restrictions are enforced by cryptography, not access control policies, sysadmins, or goodwill of providers



This makes it possible to share read-only access to a file, while retaining write access for yourself. The read-only limitation is enforced by the design of the encryption format.

Directories

- Tahoe Directories are tables, mapping childname to cap
  - table is serialized, then uploaded as a file
  - “dircap” is a filecap with instructions to interpret contents in a special way



brian takes over

Everything we’ve shown you so far has been about files: mapping a filecap to some contents. For systems where you already have a place to store the filehandles, this may be all you need. But it’s awfully convenient to use a directory structure to keep track of your filehandles.

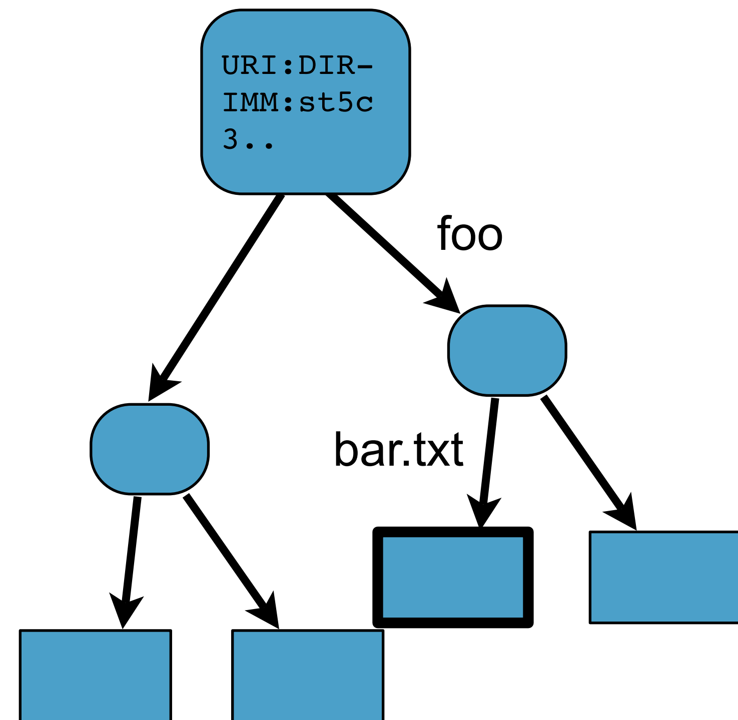
In Tahoe, we implement directories in the same capability-oriented style as we use for files. Instead of a big global tree, we manage each directory independently. In any system, a directory is just a container with a bunch of named children [diagram on left]. In Tahoe, we express this container as a table, which maps child names to their filecaps [diagram on right]. We then serialize this table into a string, and then upload the string into the grid, giving us a new filecap. Then we take the filecap for that serialized table and add a flag that marks it as a directory [box on far right]. We call this specially marked cap a **dircap**, and hold onto the dircap instead of all the original children’s filecaps. Like filecaps, dircaps are fairly short ASCII strings which can be shared through email or IM.

The Tahoe client knows how to dereference a dircap-plus-childname combination, and makes it convenient to perform the usual add-rename-delete operations on directories.

Since dircaps can be used as children too, we can nest these directories any way we like. This lets us retain an arbitrarily complex directory structure and an unlimited number of files with just a single dircap. Typically, each user manages a single such “**rootcap**”, and references everything else as paths underneath that root.

Subdirectories

- Directories can reference other directories
  - they contain dircaps and filecaps
  - lookup starts at a “root” dircap
  - then traverses one edge at a time
- $DIRCAP/foo/bar.txt

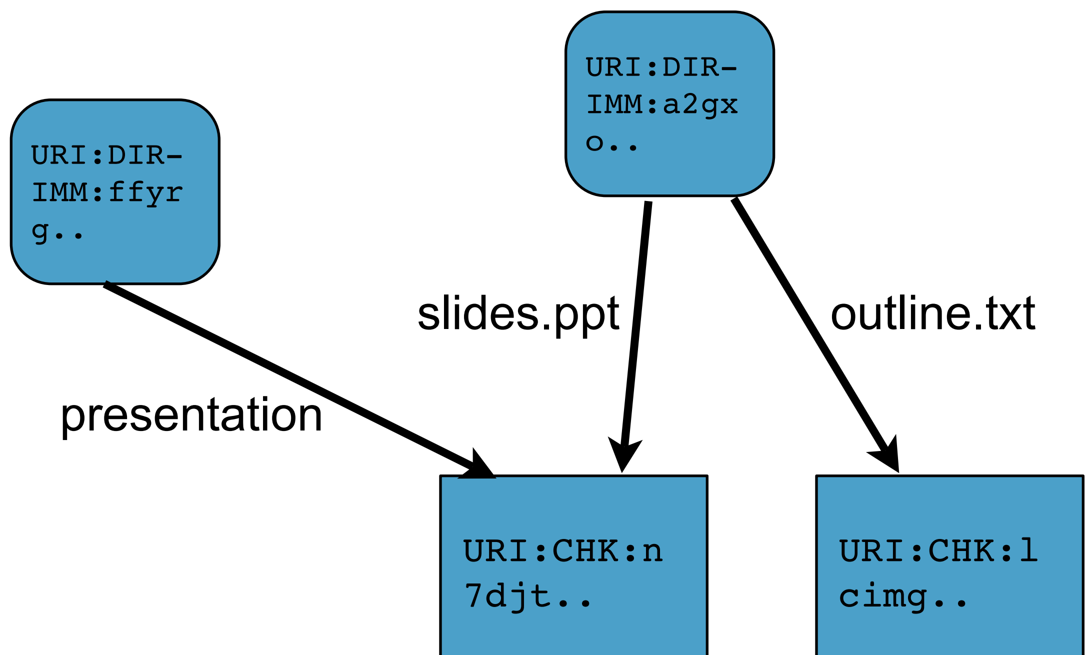


Subdirectories are merely directories that are pointed to by some other directory, like the “foo” directory in this diagram.

Note that there is no global “root directory”, merely a starting point for any particular lookup operation. Any path that you work with will always start with a dircap. The system will evaluate a path by traversing one link at a time, for each component of the path: in this example, it starts by finding the directory referenced by the dircap, then it follows the “foo” link to another directory, then it follows the “bar.txt” link to find the file we care about.

Tahoe Directory Graph

- Tahoe files and directories form a directed graph
  - names are on the edges
  - nodes are filecaps or dircaps
  - no “parent” pointers
- Files can be referenced by multiple parents

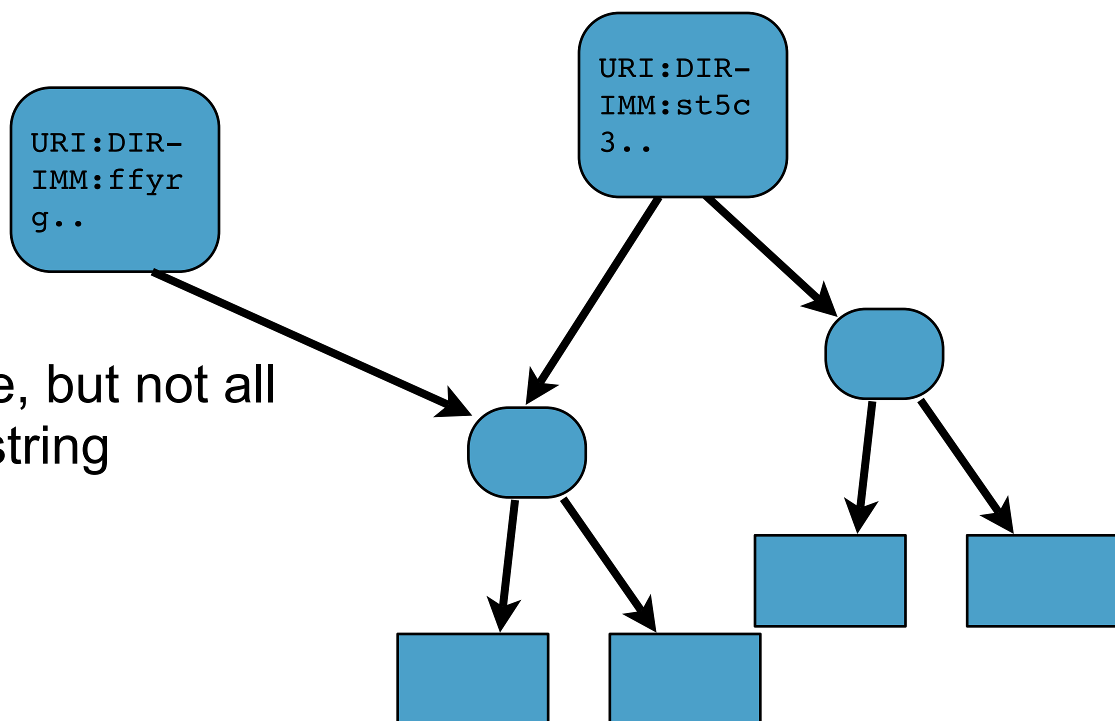


An interesting consequence of treating directories as first-class objects is that we wind up with a **directed graph** of files and directories, not a tree. Both files and directories can be referenced by multiple parents, using independent names.

One directory might reference this left-hand file under the name “presentation”, while a different directory could reference it as “slides.ppt”.

Sharing Directories

- Directories can be referenced by multiple parents
  - entire subgraphs too



  - Users can share some, but not all
  - simply pass a dircap string
  - no accounts, no ACLs

This turns out to make sharing quite easy to use and understand. By simply copy-and-pasting a single dircap, users can share arbitrarily fine-grained portions of their directory structures. There are no accounts to configure, no administrator to appeal to. Any user who can see a directory can share it.

Tahoe Directory Features

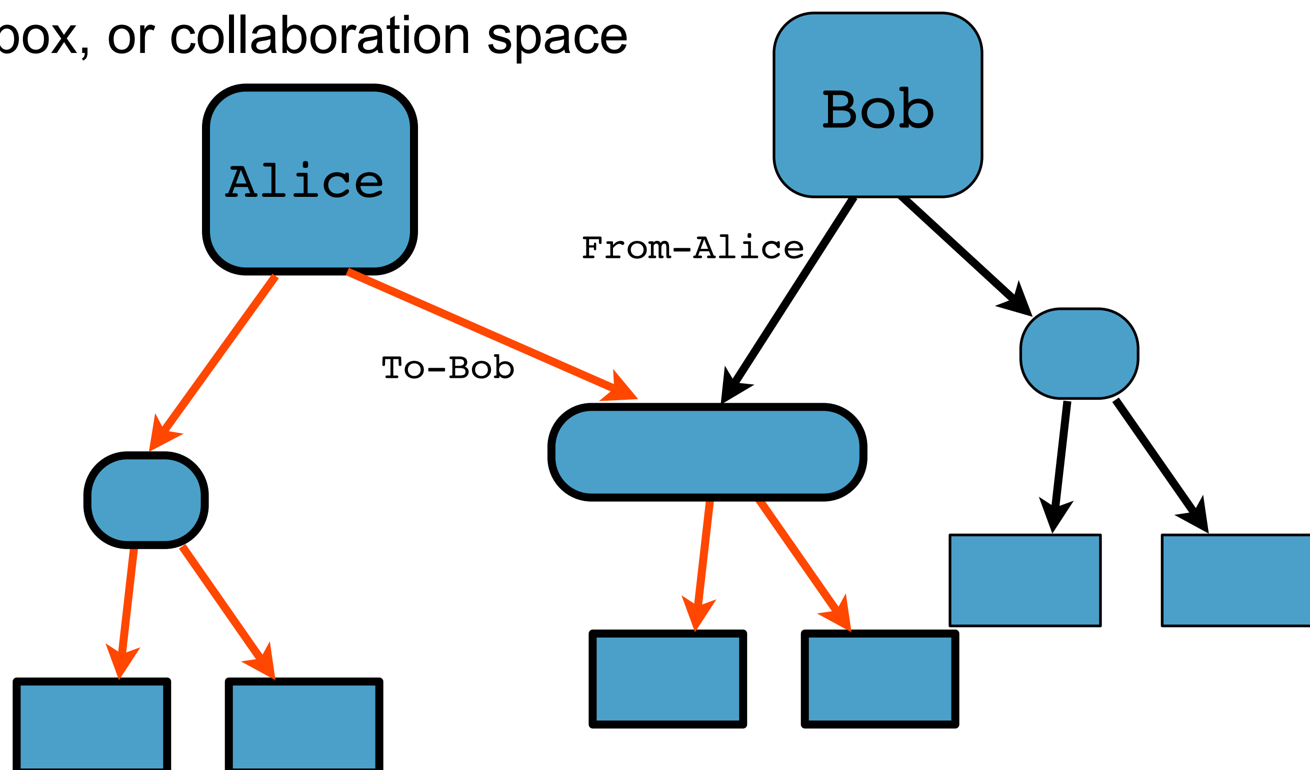
- Directories can be stored in mutable or immutable files
  - when stored in immutable, Tahoe enforces deep-immutability
- Child caps can be readcaps or writecaps
  - each directory table stores two columns: writecap, readcap
  - superencryption is used to enforce deep-readonlyness
  - all child writecaps are encrypted with a key derived from the parent writecap before encoding
- Users can grant read-only access to a directory subtree
  - while retaining write access for themselves or others

Tahoe directories can be mutable or immutable, depending upon what kind of file they're stored in. For the mutable ones, there are both readcaps and writecaps. These properties are "deep" (i.e. transitive): everything you can reach through a directory **read**cap will be read-only, and everything you get through an **immutable** dircap will also be immutable.

This makes it easy to grant read-only access to a subtree, while retaining write access for yourself.

Sharing Directories

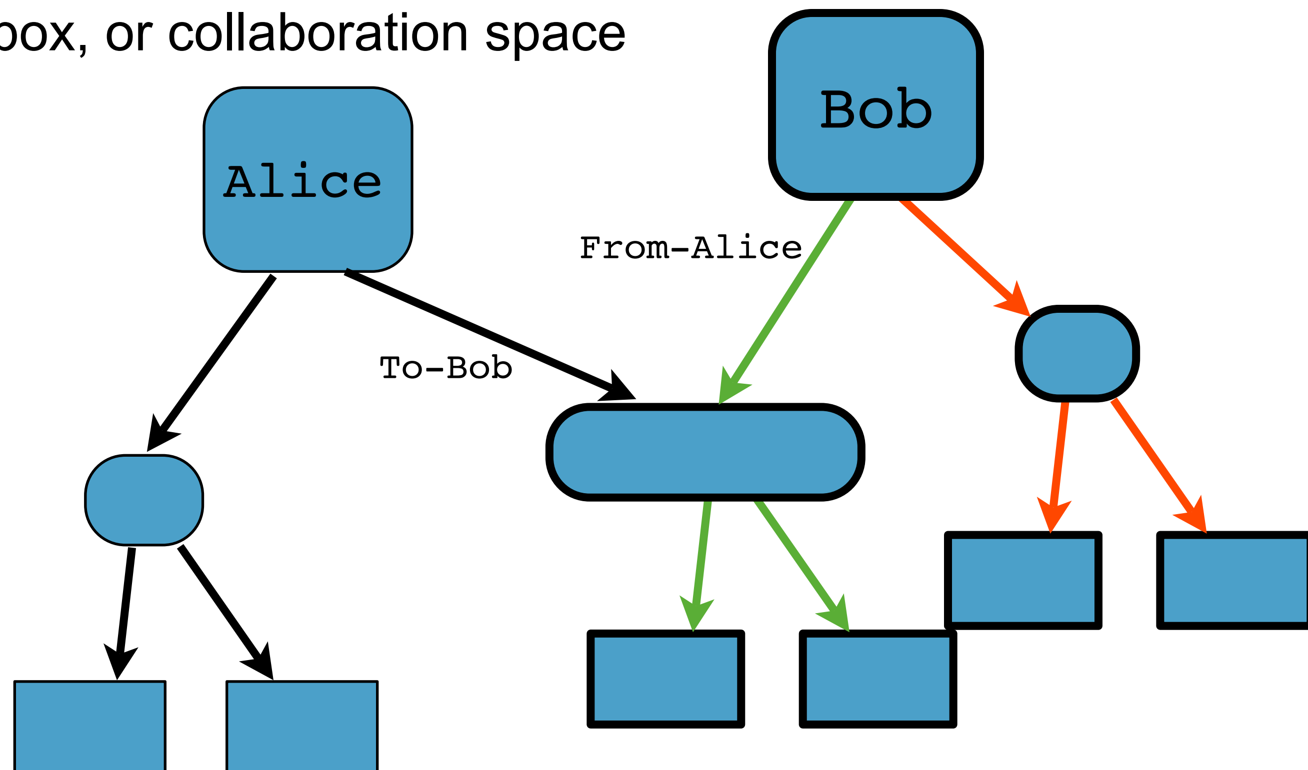
- Users can share a common directory
  - use as a mailbox, or collaboration space



Tahoe users routinely link a shared subdirectory underneath their personal rootcap, and use it to exchange files with a collaborator. For example, Alice can create an “outbox” for Bob, by creating a new subdirectory and handing him a readcap to it. Bob then links the readcap into his own directory space, under a name of his choice. Alice will be able to write into that directory (as indicated by the red arrows). But Bob, through his readcap..

Sharing Directories

- Users can share a common directory
  - use as a mailbox, or collaboration space



.. will only be able to read the shared files (in green), not modify them. The type of access (readonly, read-write) is determined solely by the links through which each user traverses.

Nobody else get any access, unless Alice or Bob specifically grants it by passing them the dircap.

[if there is time, maybe show a demo of dir-readcaps in the web-ui, showing what this “To-Bob” directory looks like from both sides]

POLA

Principles Of Least Authority

- Share a single file or directory, not all your files
- Share readonly access, not read+write
- No component of the system (gateway, storage server) has more authority than it needs
  - compromised storage servers cannot violate security
  - compromised gateways can only violate security of filecaps you give them
  - no “root”, no administrators with extra powers
- Break up your design into pieces, give each piece least authority

We’ve designed Tahoe to express the Principle of Least Authority at many levels. From the user’s point of view, the filesystem allows easy, fine-grained sharing of specific files and directories. We believe that sharing filecaps and dircaps is easier than persuading an administrator to update a table of ACLs. And as we all know, if users cannot easily accomplish their goals within the system, they’ll accomplish them outside the system, which means just sharing their passwords with each other. That would be the Principle Of Maximum Authority, and that’s bad.

We believe that using filecaps and dircaps makes it easier for users to visualize the scope and scale of the authority they’re sharing, more so than hidden control panels and properties dialog boxes.

At the lower levels, the cryptographic protocols ensure that only the user’s gateway can see or change the user’s files, and the storage servers are not used for anything but storing ciphertext.

As a general design principle, we strongly encourage application developers to follow this approach: break your design up into pieces, give each piece as little authority as possible. Not only does this improve security, it helps debugging, maintenance, reliability against bitrot, and code readability.

Verifycaps

- All files have a “verifycap”
- contains integrity-checking hashes, storage-index
  - but *no* decryption keys
- verifycaps can be used to check integrity of ciphertext
  - allows servers, other non-trusted parties to do maintenance work
- new shares (for existing files) can be created using just the verifycap
  - allows non-trusted parties to perform repair work
- lets you take advantage of machines that would normally be off-limits due to security considerations

skip if short on time

Another expression of POLA is the Tahoe **verifycap.** Each file has one, and it’s a string that holds the integrity-checking hashes, but **not** the decryption key. If you have one of these, you can find the shares, verify every single bit against corruption, reconstruct the ciphertext, even generate new shares that are guaranteed to be bit-for-bit identical to the original ones, but **not** recover the plaintext.

This lets you safely delegate data scrubbing and repair work to anyone you like: volunteers, a paid repairer service, even the storage servers themselves can participate in the job of keeping those shares healthy. This lowers your maintenance costs by expanding the pool of eligible worker machines.

Other Tahoe-LAFS Features

- Garbage Collection
  - leases on shares, updated periodically, shares expire
- Web Browser -oriented UI
  - Server management, grid status, current activity, performance
  - provisioning/reliability calculation tools
- Verify/Repair
  - scan shares for errors, replace corrupted ones
  - deep traversal from a starting dircap
- Good Alacrity
  - use of Merkle Hash Trees for verification
  - fetch minimal data (128KB) before returning plaintext

Some other Tahoe features that you might want to learn more about later: we have a lease-based garbage collector to allow servers to delete shares for files that are no longer in use, to recover disk space. There is a browser-based UI which includes some grid management/status tools. There are both web and CLI based tools to perform deep traversal of a directory structure and look for problems like missing or corrupted shares, and to automatically repair any damage that's found.

And finally, Tahoe maintains a goal of providing low-alacrity access to files, meaning that no matter how large the file is, you should be able to start streaming a download quickly, without fetching very much data from the servers. By using a Merkle hash tree for integrity protection, we never have to retrieve more than about 128KB of data to start producing validated plaintext.

[maybe do demo here of streaming playback of a large movie file, to show off low alacrity]

Ongoing Work

- Smaller filecaps, Faster mutable files
  - RSA keypair generation takes a second or two
  - ECDSA would take milliseconds
  - new formats for shorter filecaps
- Accounting
  - tracking+limiting how much space is consumed by each user
- More frontends
  - WebDAV, Browser plugins
  - tahoe://filecap -style URLs

Some of the future projects we're working on include new cryptographic formats, using elliptic curve DSA, to get smaller filecaps and faster creation time for mutable files. We're working on an accounting system to help keep track of how much server space is consumed by each user, so they can be billed or limited appropriately. And we're working on more frontend protocols, in particular a WebDAV frontend would get us easy integration with all the common operating systems.

Related Projects

- Hadoop-lafs
- Duplicity backup plugin
- TiddlyWiki in Tahoe-LAFS
- Android, iPhone clients

zooko concludes

Taking It Home

- Analyze your current storage architecture to determine what components you rely upon for **confidentiality**, **integrity**, and **availability**
- **Install** a Tahoe-LAFS grid, use it to store and share data
- Use these same techniques to build a storage system that gets you **provider-independent security**
- Don't rely upon your cloud storage provider for security: **Bring Your Own Security**

So hopefully, armed with today's knowledge, here's what you can do when you get home.

More Info?

http://tahoe-lafs.org

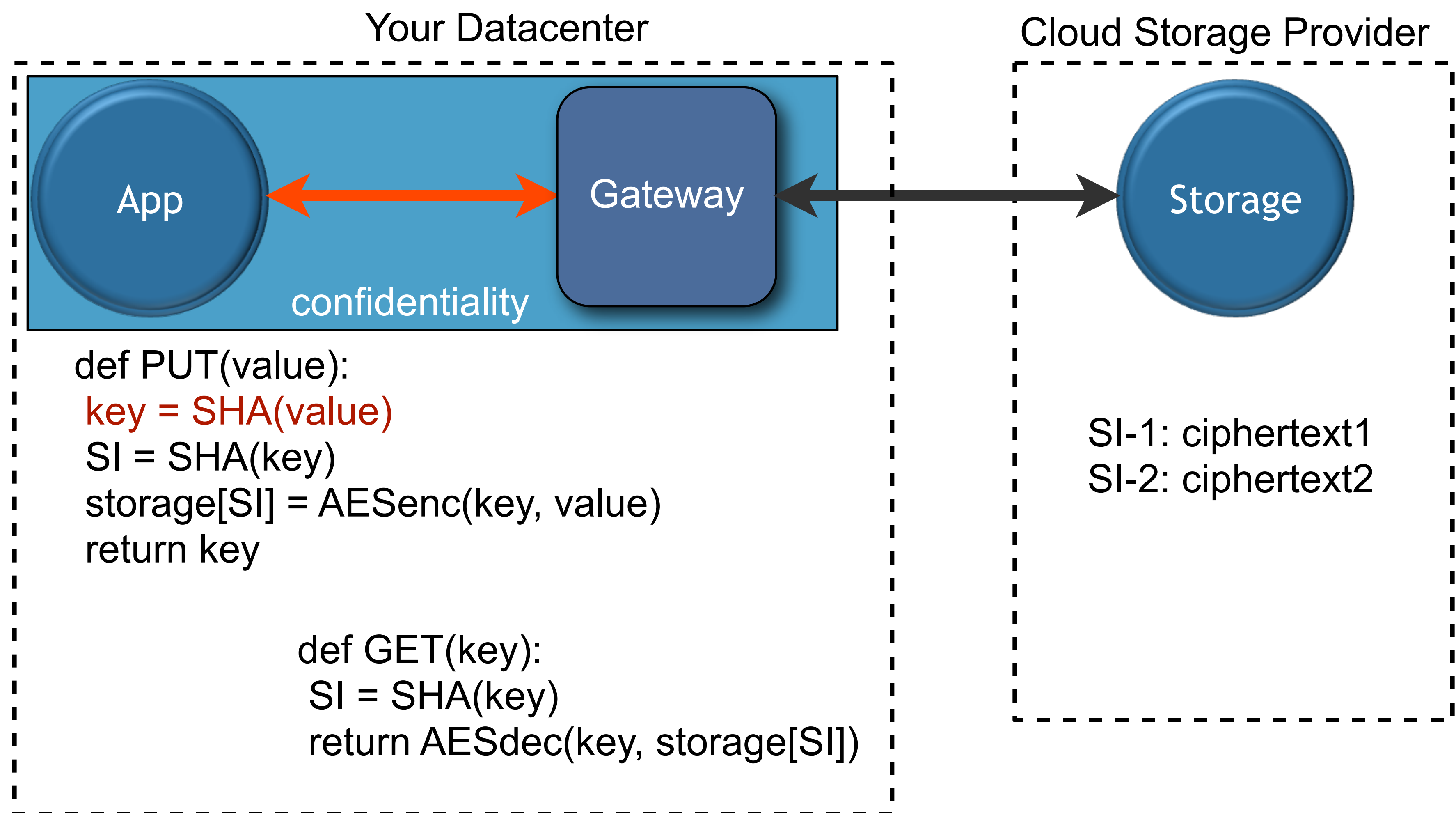
tahoe-dev@allmydata.org

Audience Participation Demo:
http://tahoe-lafs.org/RSA

source code, installation instructions, bug tracker, mailing list, IRC channel

bonus slides

Convergent Encryption



Optionally, we use another trick called “convergent encryption”, in which the encryption key is a secure hash of the plaintext. This has the convenient property that uploading the same file twice results in the same ciphertext, which can be shared between the two instances to save space.

This doesn’t affect the GET code at all.