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

RSACONFERENCE2010

SECURITY DECODED

Brian Warner
Zooko Wilcox-O’Hearn

Session ID: AND-402
Session Classification: Advanced

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 Stored in a Cloud
- Your Right to Security
- Better Options
- How Tahoe-LAFS Implements Those Options

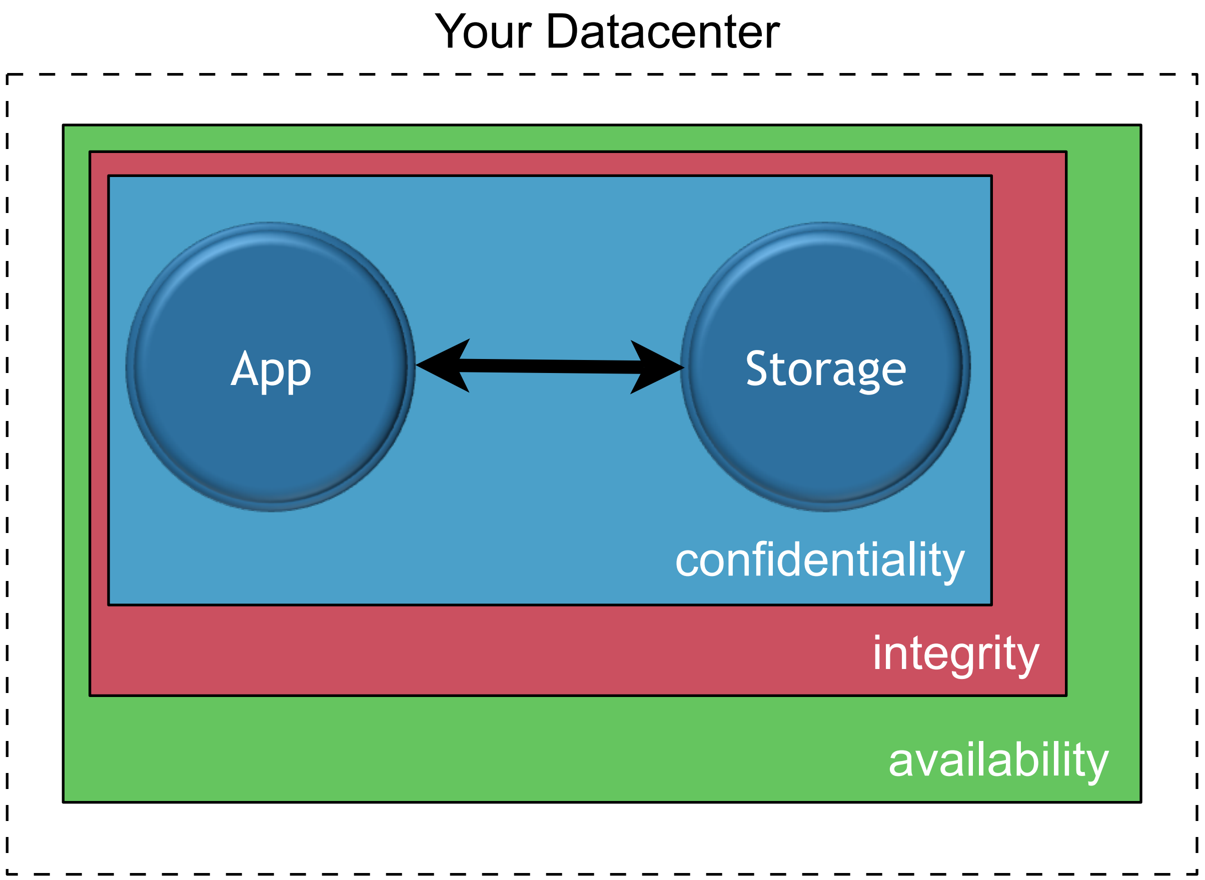
What We Want You To Take Home

- Beliefs:
  - You deserve confidentiality and integrity even when you buy reliability and availability from a cloud storage provider
  - Tahoe-LAFS is an open-source system which provides good properties
- Skills:
  - Identify which properties rely upon which components
  - Install and use a Tahoe-LAFS storage grid
- Tools:
  - Decorrelate failures
  - Erasure coding provides tunable reliability-vs-overhead, better than straight replication

Cloud Storage is Cheap, Easy, and Scalable

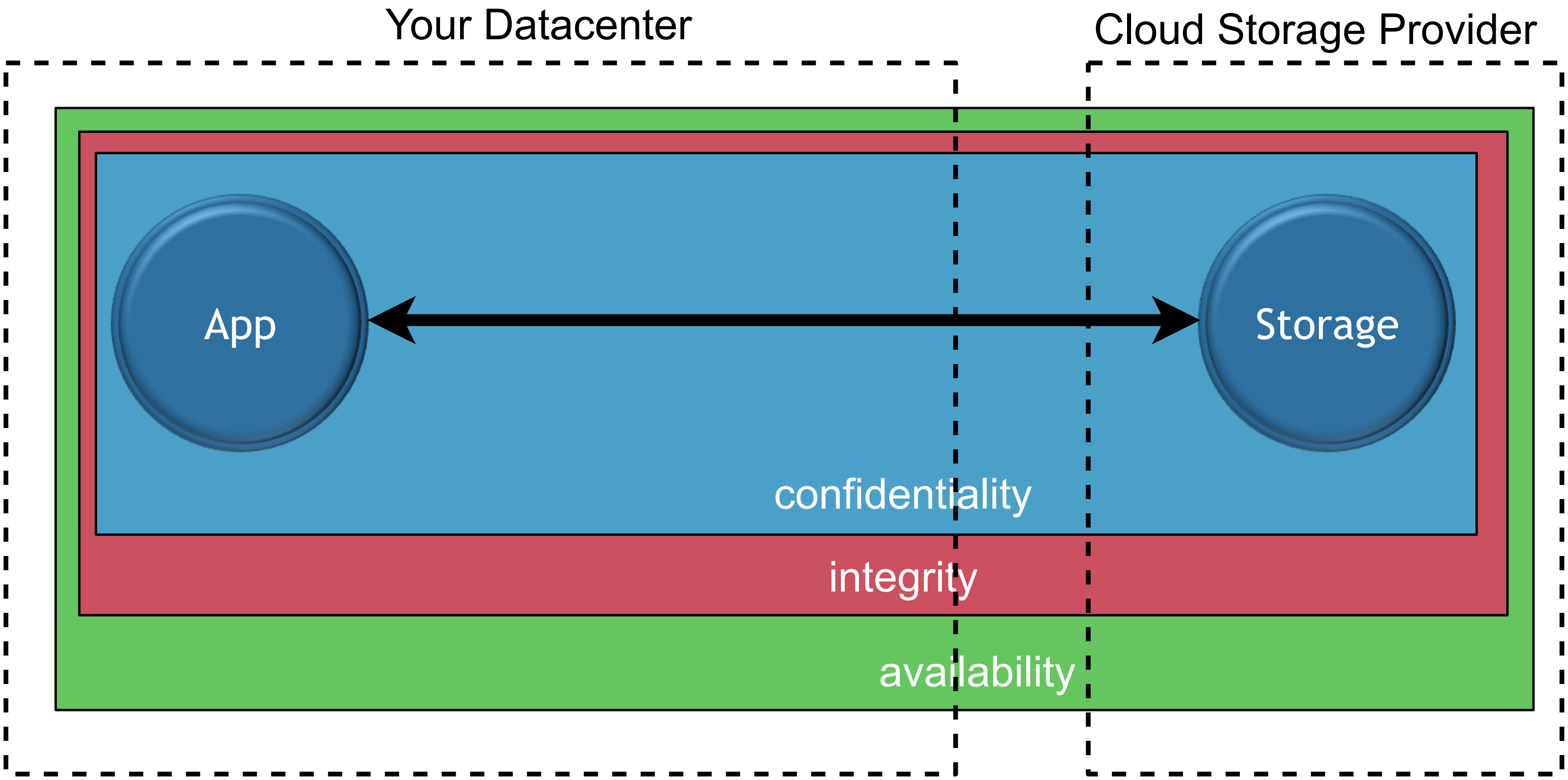
- plenty of vendors: Amazon, Rackspace, Google
- but it changes the security story
  - who else can see your data?
  - who else can modify your data?

Property Perimeters



Everyone understands the notion of a security perimeter. To maintain security, everything inside the perimeter must function as expected. We can refine this to talk about separate perimeters for separate properties. In the case of storage, we care about confidentiality, integrity, and availability. For data that you manage on your own hardware, you get these properties if all of your own hardware works correctly and remains uncompromised. This is hard enough: any admin in your organization could mess up, any of your machines could fail, or someone might break in.

Drawing Perimeters Around Clouds



Your Datacenter

Cloud Storage Provider

App

Storage

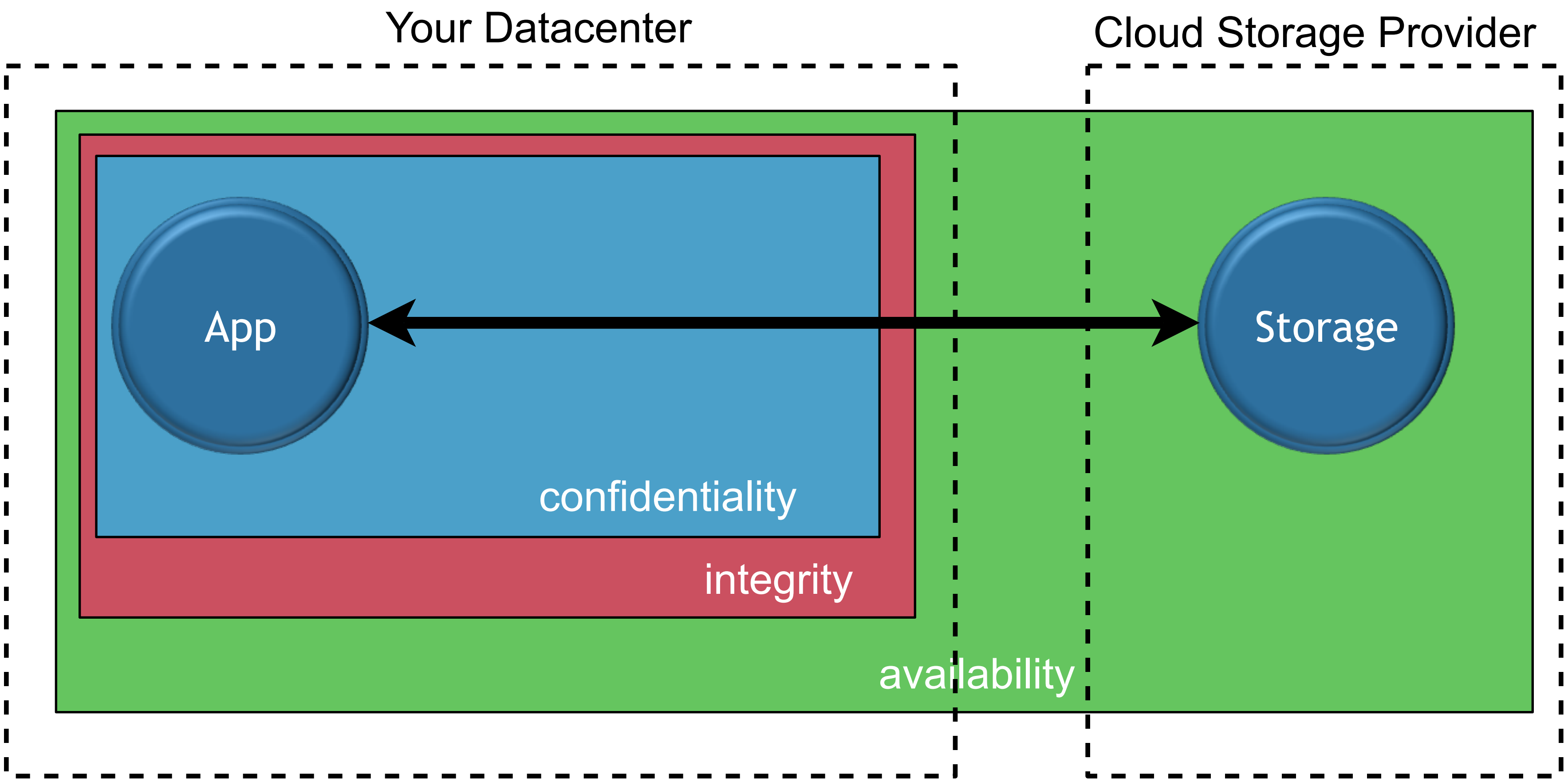
confidentiality

integrity

availability

Including outsourced storage stretches all the perimeters. 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? What sorts of assurances can you have? This is an economic tradeoff, but made with hardly any information.

Separate the Perimeters

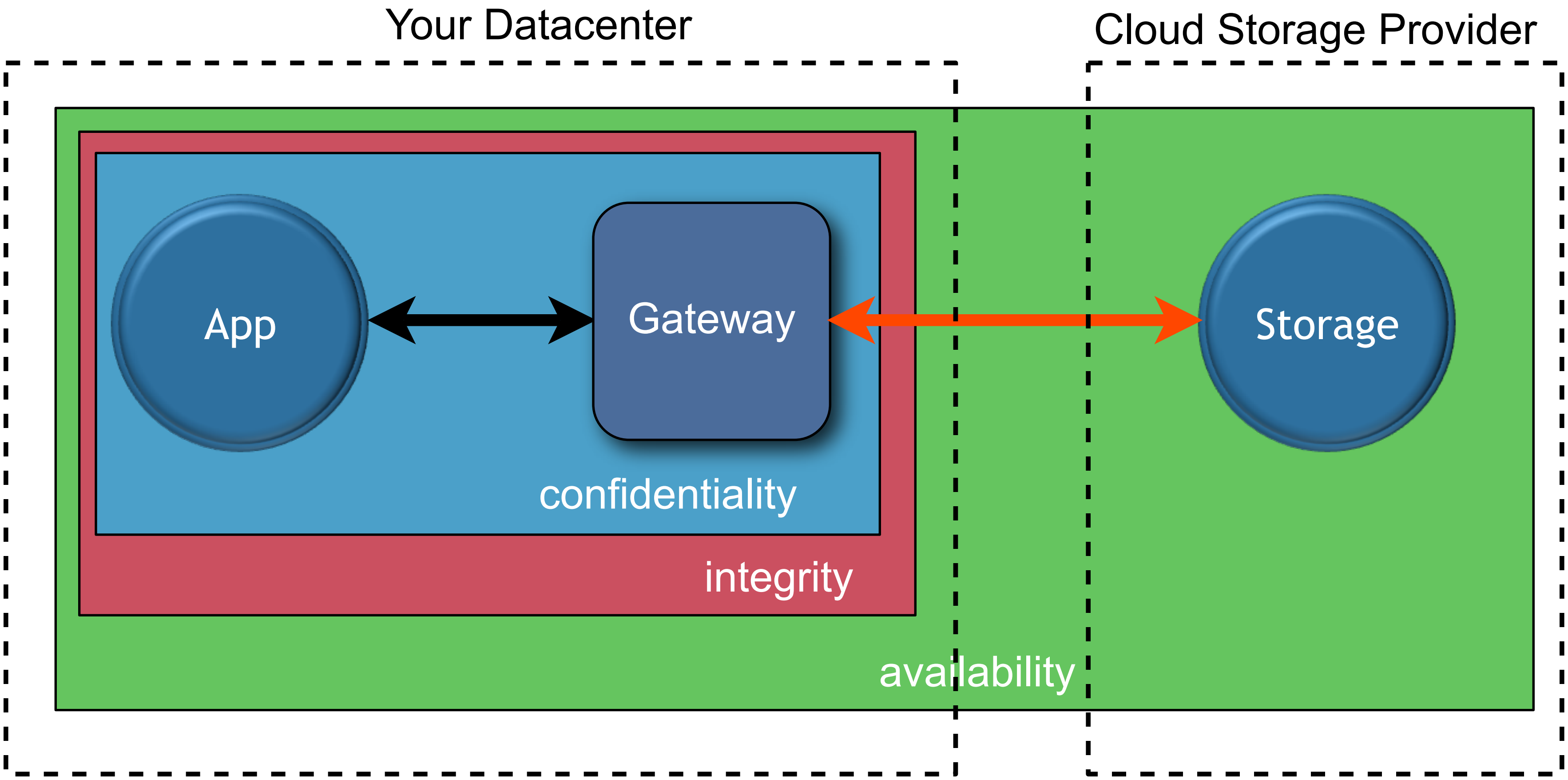


So our goal is to separate these concerns. Buy availability from your storage provider, but bring your own security. Measuring availability, while not easy, is far simpler than measuring the security they claim to provide you.

You know your system works when your confidentiality is not breached even if your storage provider publishes anything you give them to the whole internet. Likewise, if you can detect even a single bit flip in your provider’s storage, then your integrity will not be breached (even though your availability may suffer).

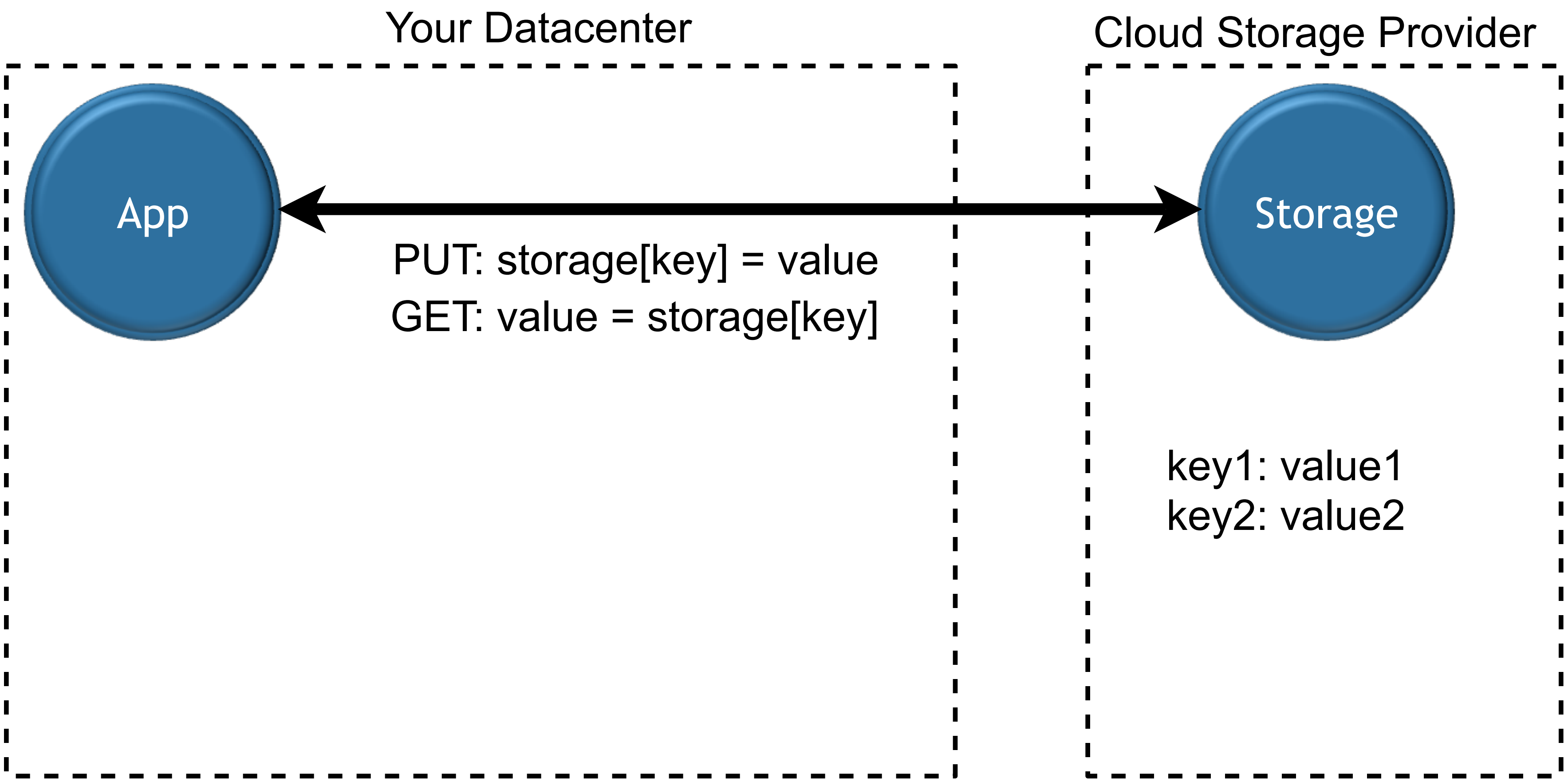
Now you’re making a cost-benefit analysis based upon maintaining availability alone, which is a much easier decision to manage.

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 and the validated ciphertext that you give to your storage provider.

Key-Value Store



Your Datacenter

Cloud Storage Provider

App

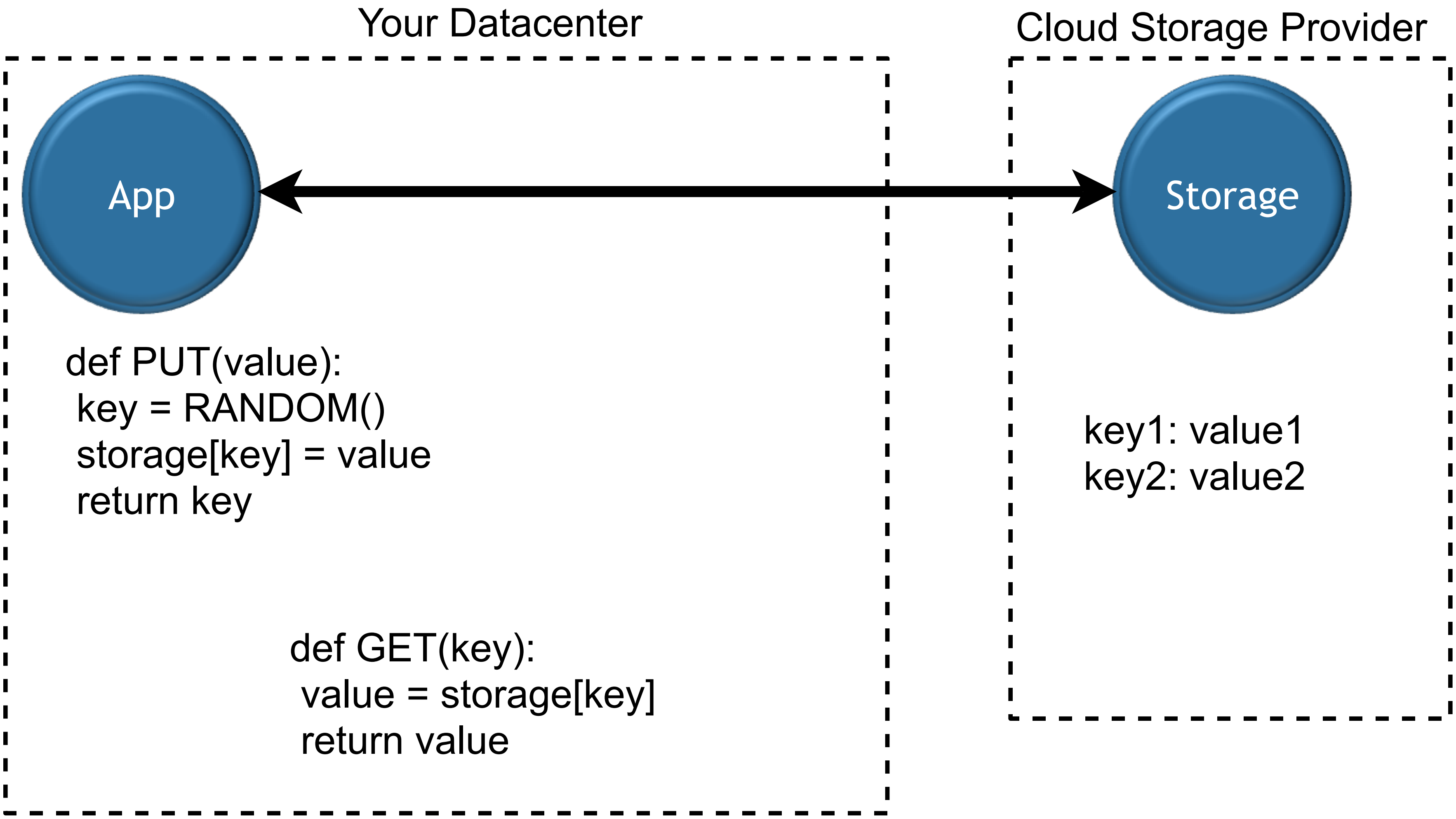
Storage

PUT: storage[key] = value
GET: value = storage[key]

key1: value1
key2: value2

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. For many applications, the key is an opaque unique string, and is frequently stored in some other data structure like a database "foreign key" column.

Opaque Key-Value Store



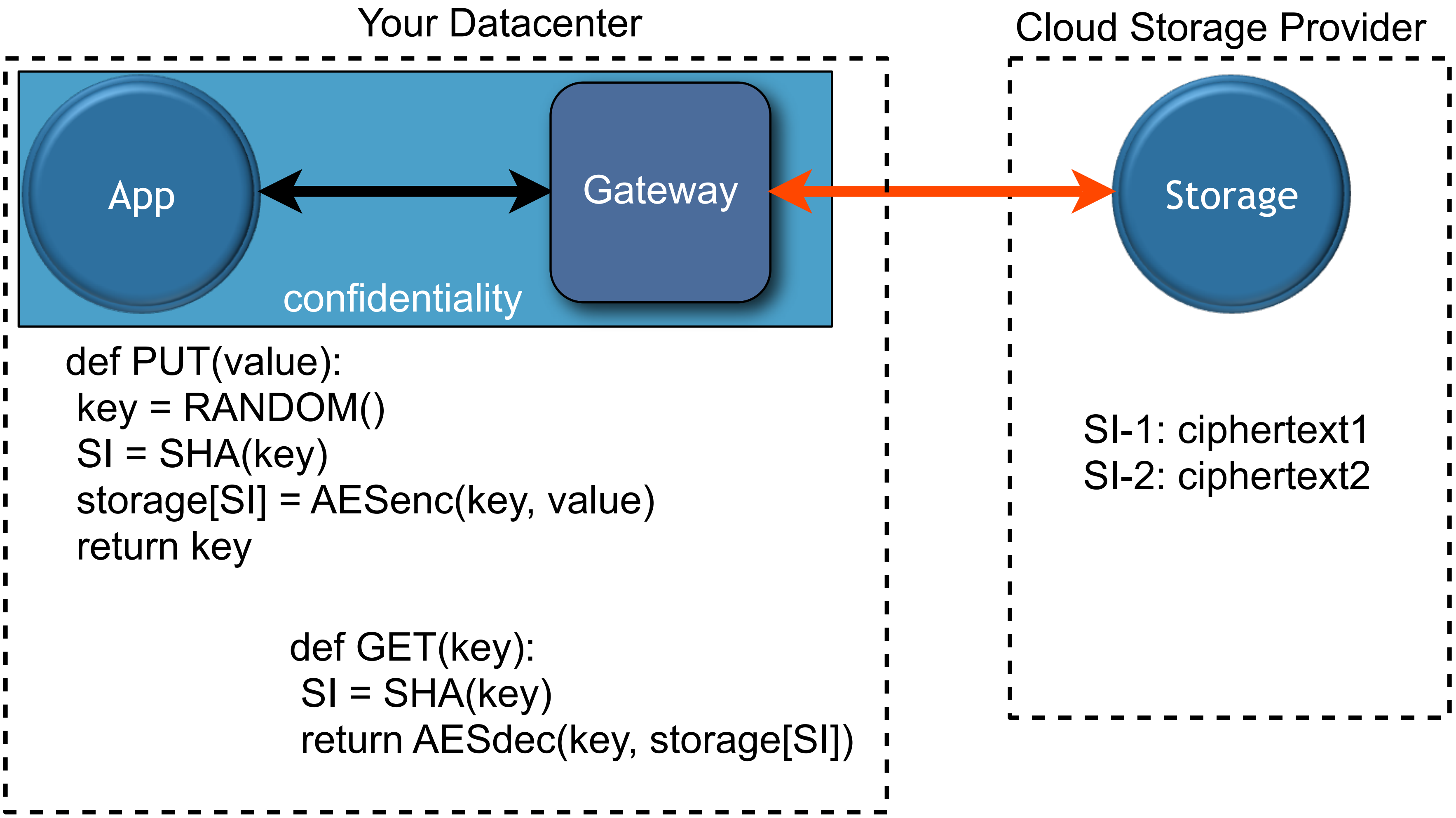
```
def PUT(value):
 key = RANDOM()
 storage[key] = value
 return key
```

```
def GET(key):
 value = storage[key]
 return value
```

key1: value1
key2: value2

For many applications, the key is an opaque unique string, and is frequently stored in some other data structure like a database “foreign key” column. You can think of this as a file-handle for this particular piece of data.

Encrypt Before Store



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

```
def GET(key):
 SI = SHA(key)
 return AESdec(key, storage[SI])
```

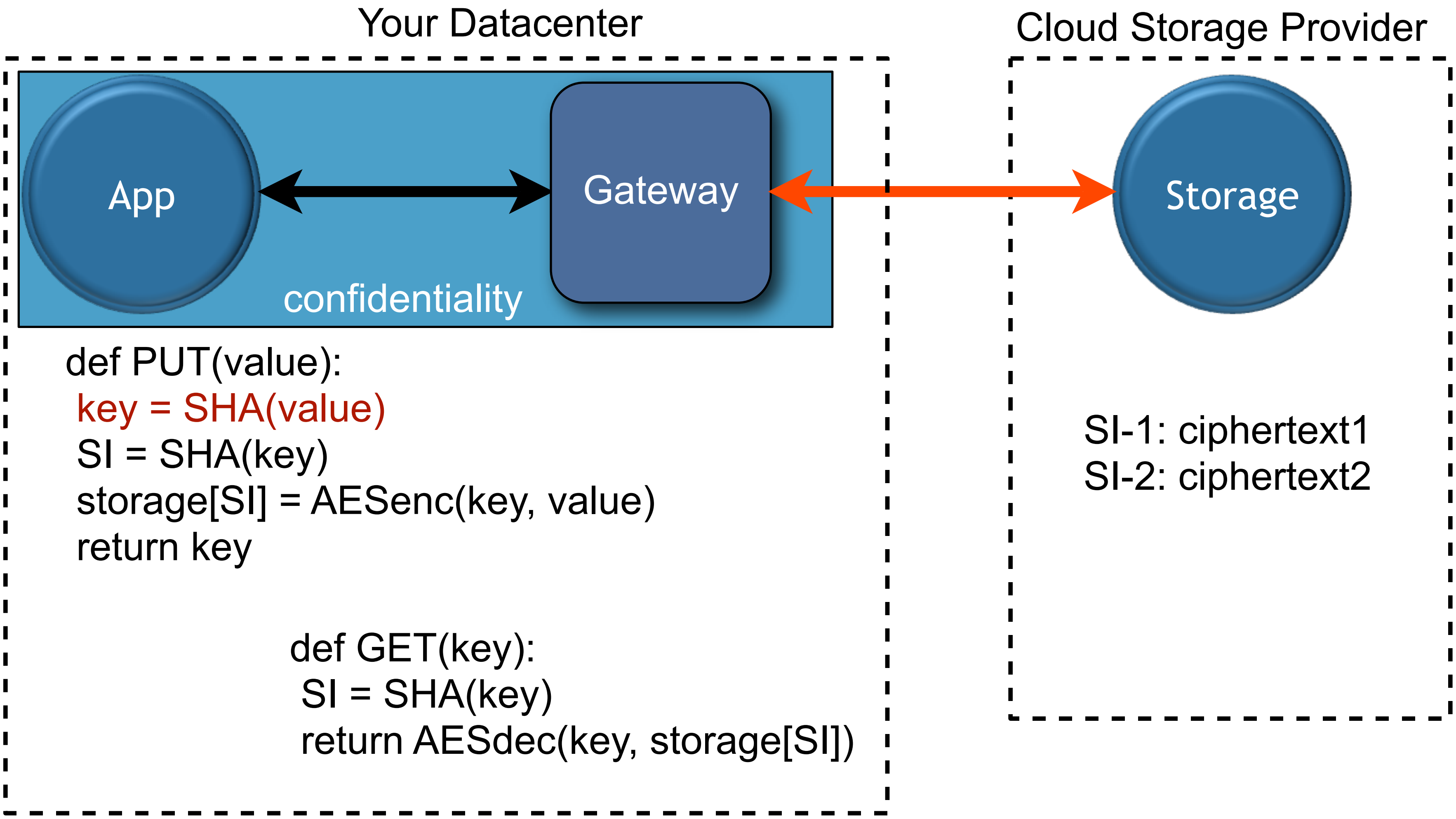
SI-1: ciphertext1
SI-2: ciphertext2

Now we build a gateway which encrypts the data before giving it to the storage system. We generate a random AES key for each file, and return it to the application as the filehandle. We use a one-way hash to derive a “storage index”, with which we tell the cloud where to store our 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.

This removes the storage system from the confidentiality perimeter. Nothing the storage host can do will compromise the confidentiality of our data. 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.

Convergent Encryption



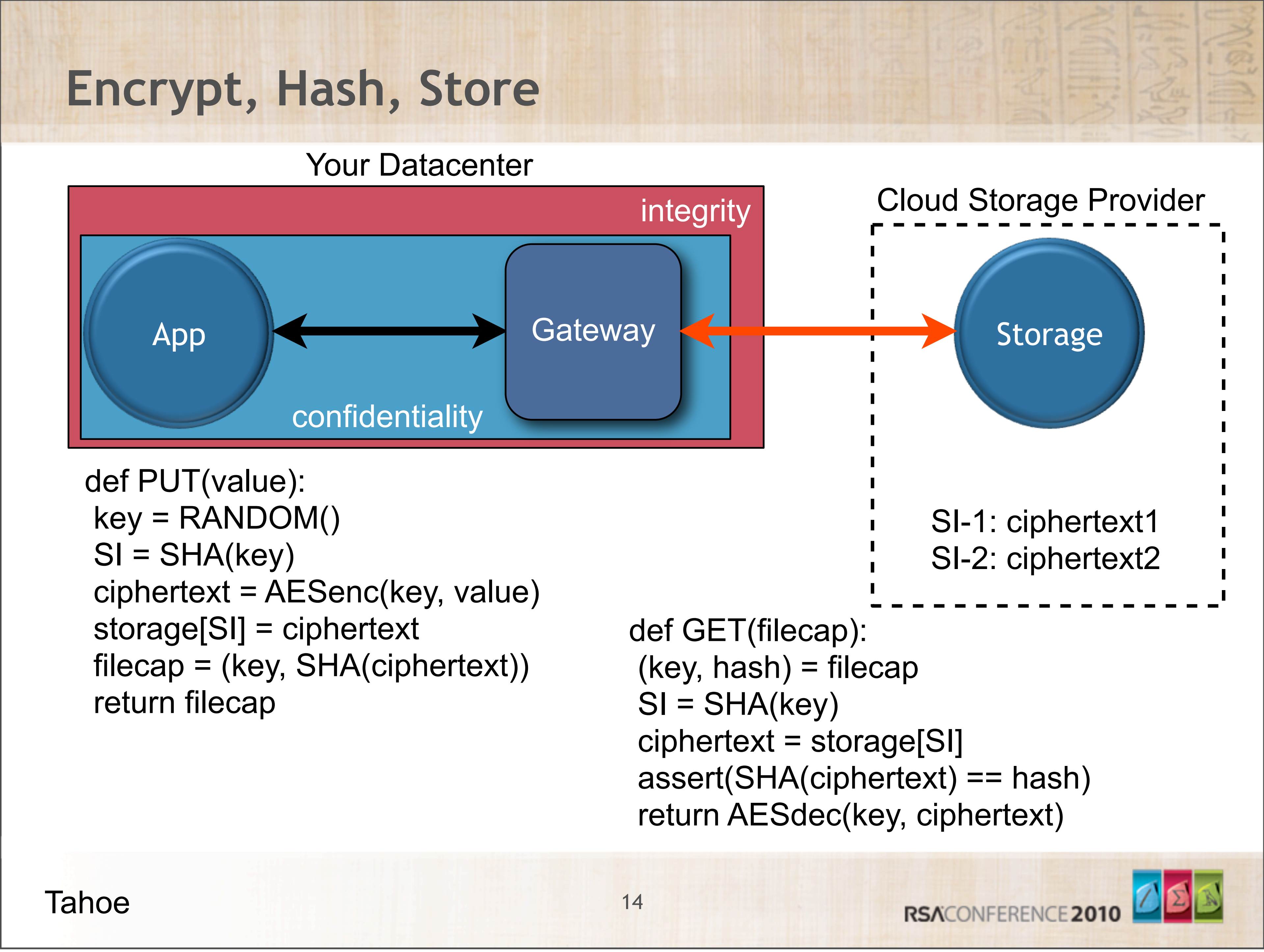
```
def PUT(value):
  key = SHA(value)
  SI = SHA(key)
  storage[SI] = AESenc(key, value)
  return key
```

```
def GET(key):
  SI = SHA(key)
  return AESdec(key, storage[SI])
```

SI-1: ciphertext1
SI-2: ciphertext2

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.



Encrypt, Hash, Store

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

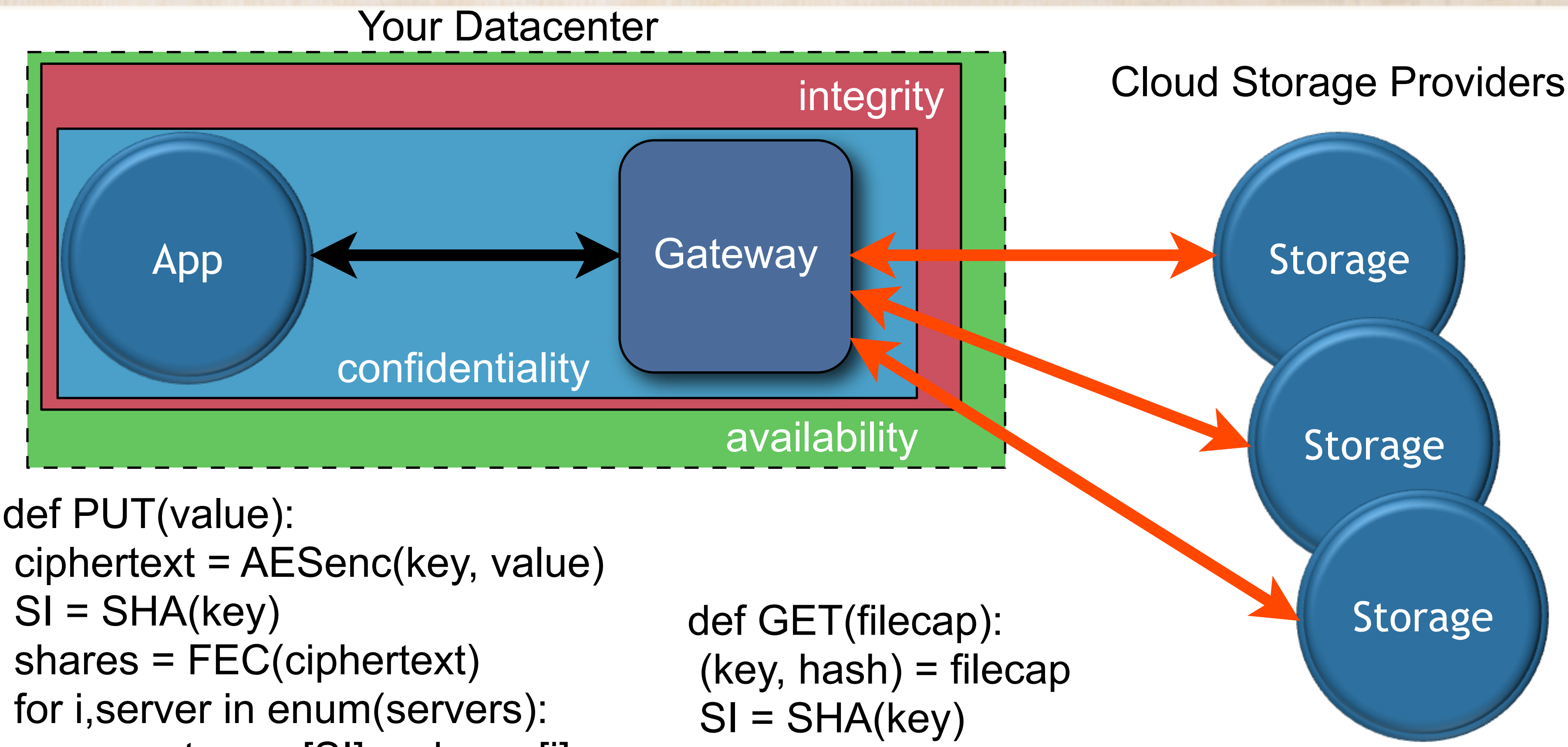
```
def GET(filecap):
 (key, hash) = filecap
 SI = SHA(key)
 ciphertext = storage[SI]
 assert(SHA(ciphertext) == hash)
 return AESdec(key, ciphertext)
```

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. 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) to allow untrusted parties to participate in verification: given a storage-index and a hash, anyone who can access the storage service can verify that the data is intact.

You’d actually want to use Merkle trees here, so you can check integrity on smaller pieces, without first having to download the entire file.

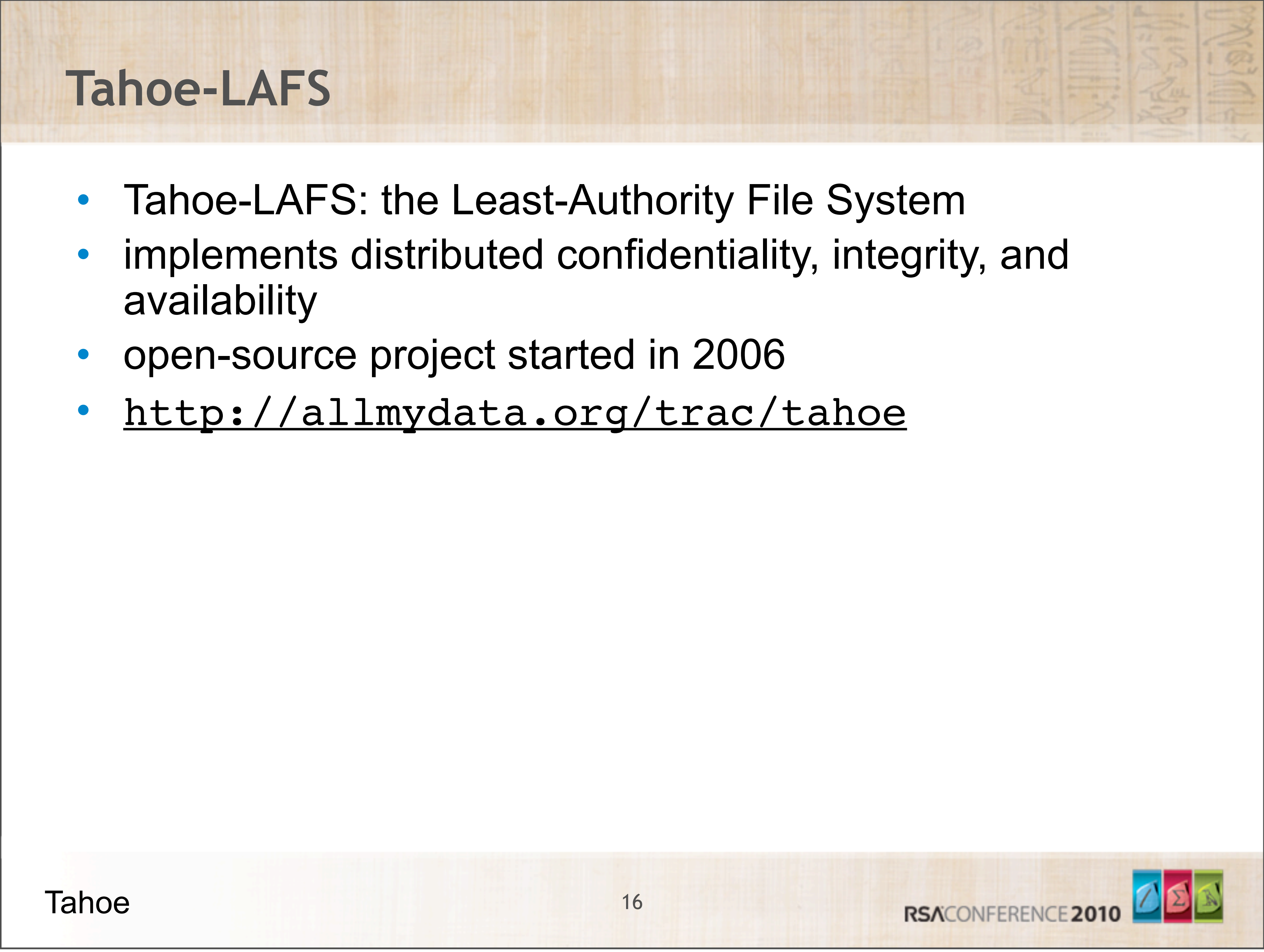
Erasure Coding for Reliability



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

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

and for extra credit, we can apply erasure coding, or 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. We can send each piece to a different provider, 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 (and less available) servers, or it might let you achieve a higher availability goal than any one provider can offer. The tradeoff between cost and quality is decided by your gateway.

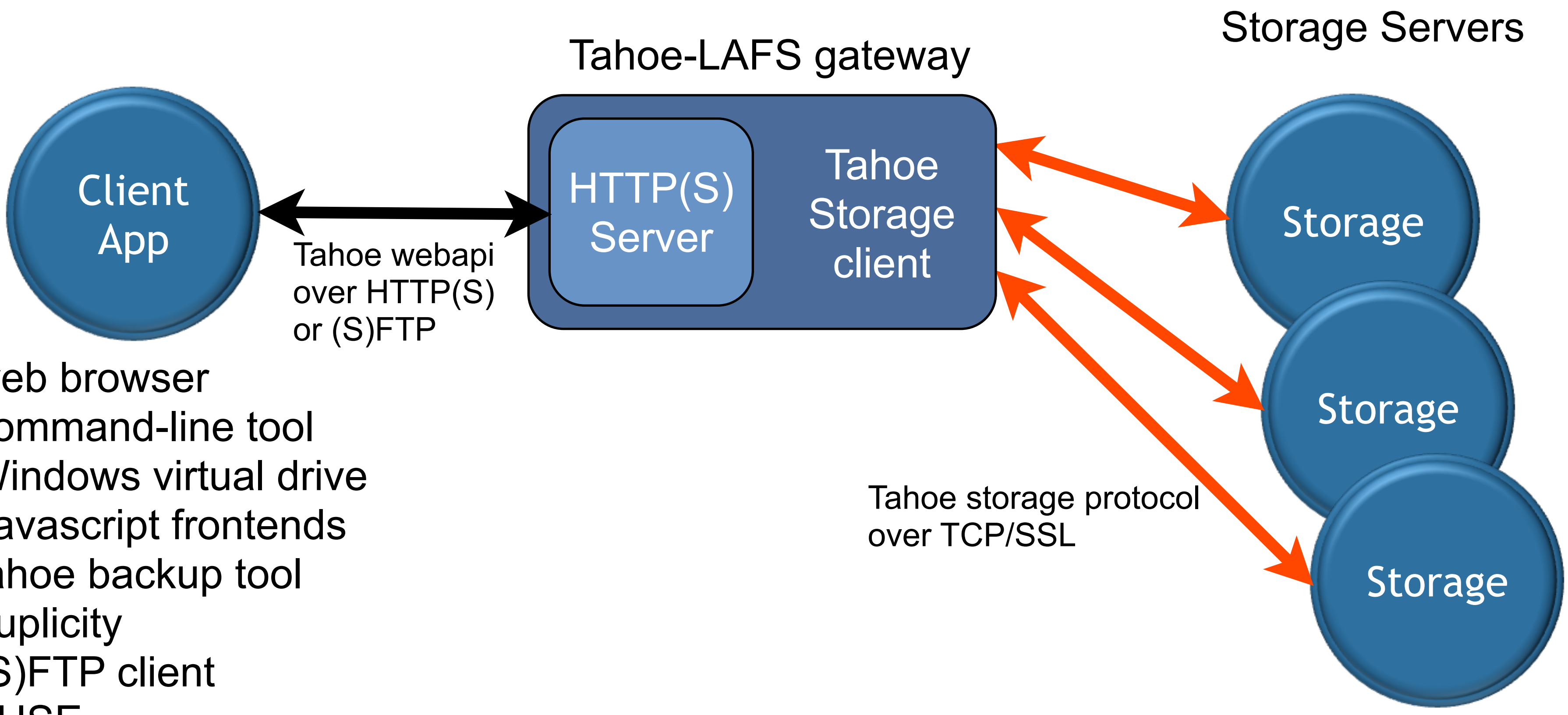


Tahoe-LAFS

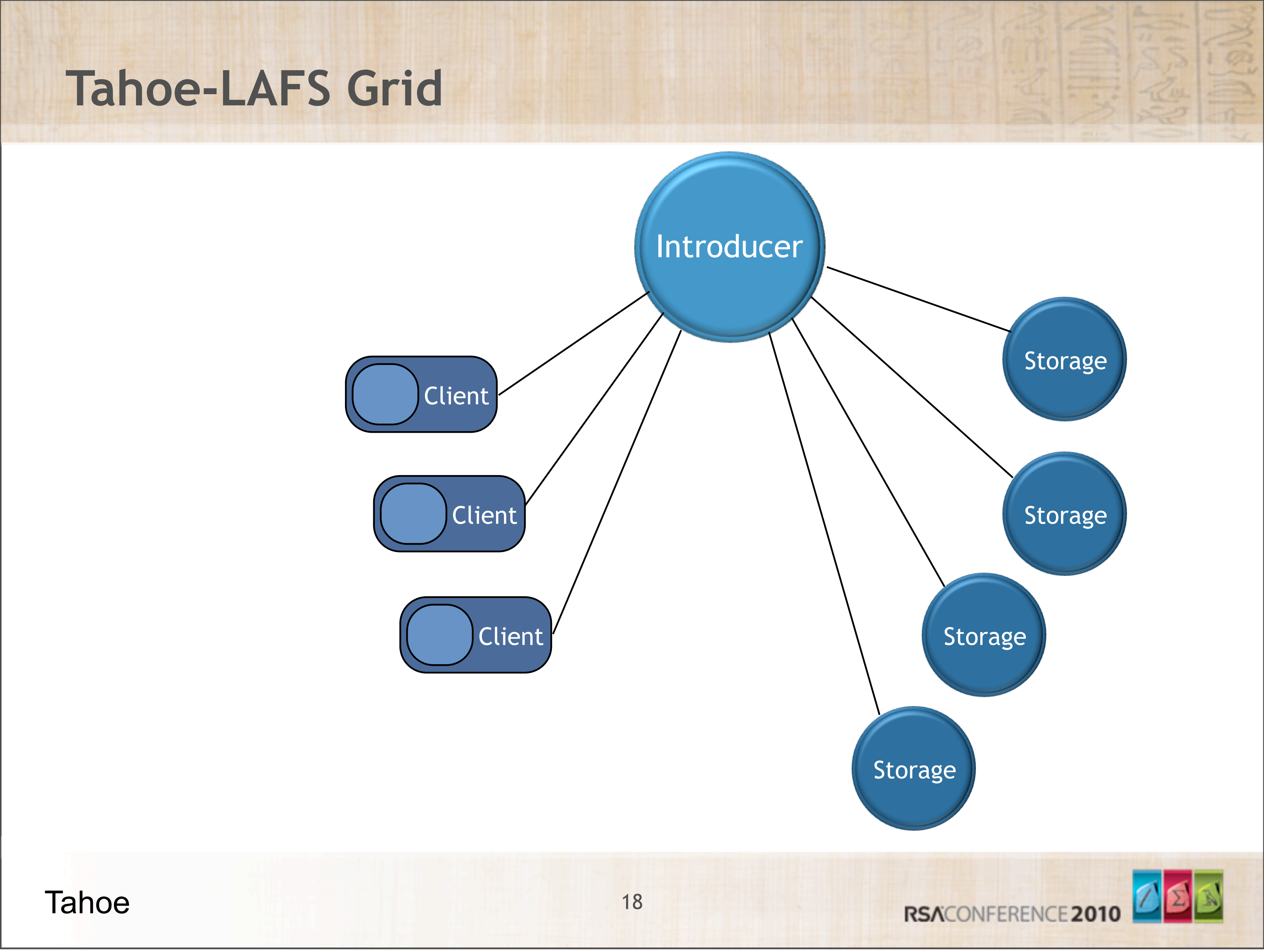
- Tahoe-LAFS: the Least-Authority File System
- implements distributed confidentiality, integrity, and availability
- open-source project started in 2006
- `http://allmydata.org/trac/tahoe`

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: Overview



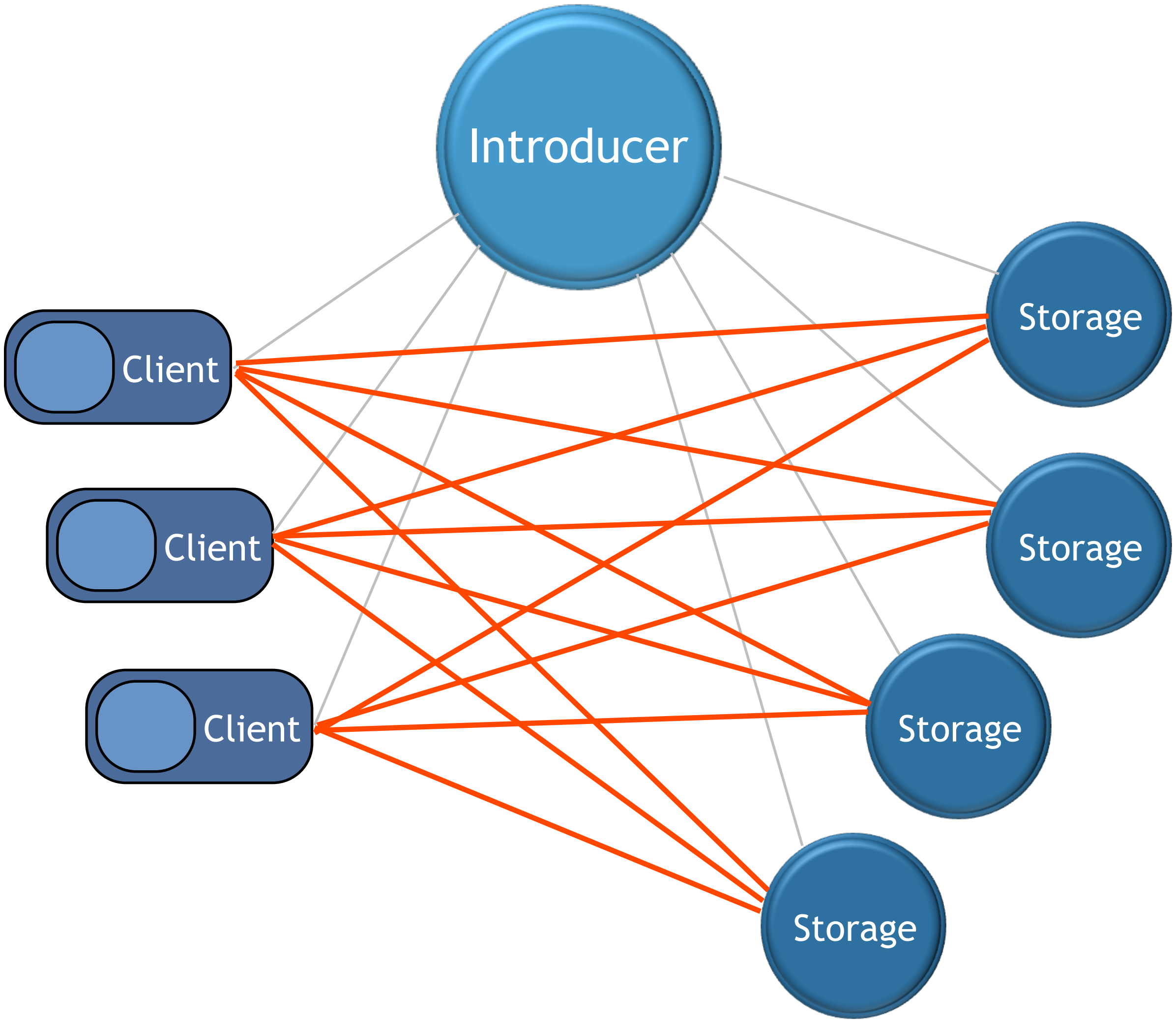
- web browser
- command-line tool
- Windows virtual drive
- Javascript frontends
- tahoe backup tool
- duplicity
- (S)FTP client
- FUSE



Tahoe-LAFS Grid

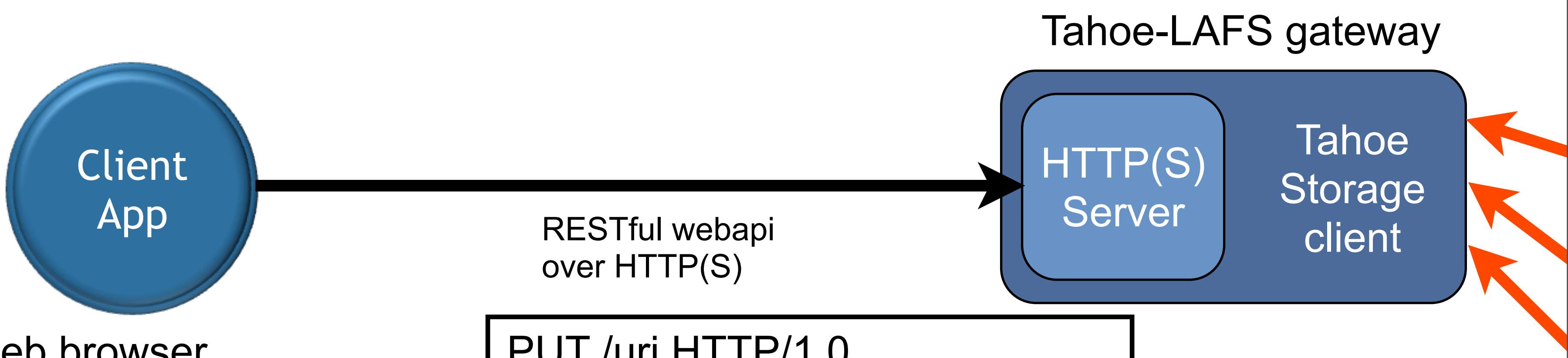
The Tahoe grid is established by means of an “Introducer”. All nodes connect to the introducer, both clients and storage servers. The Introducer distributes location information about all other nodes, allowing..

Tahoe-LAFS Grid



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

Tahoe CLI, webapi

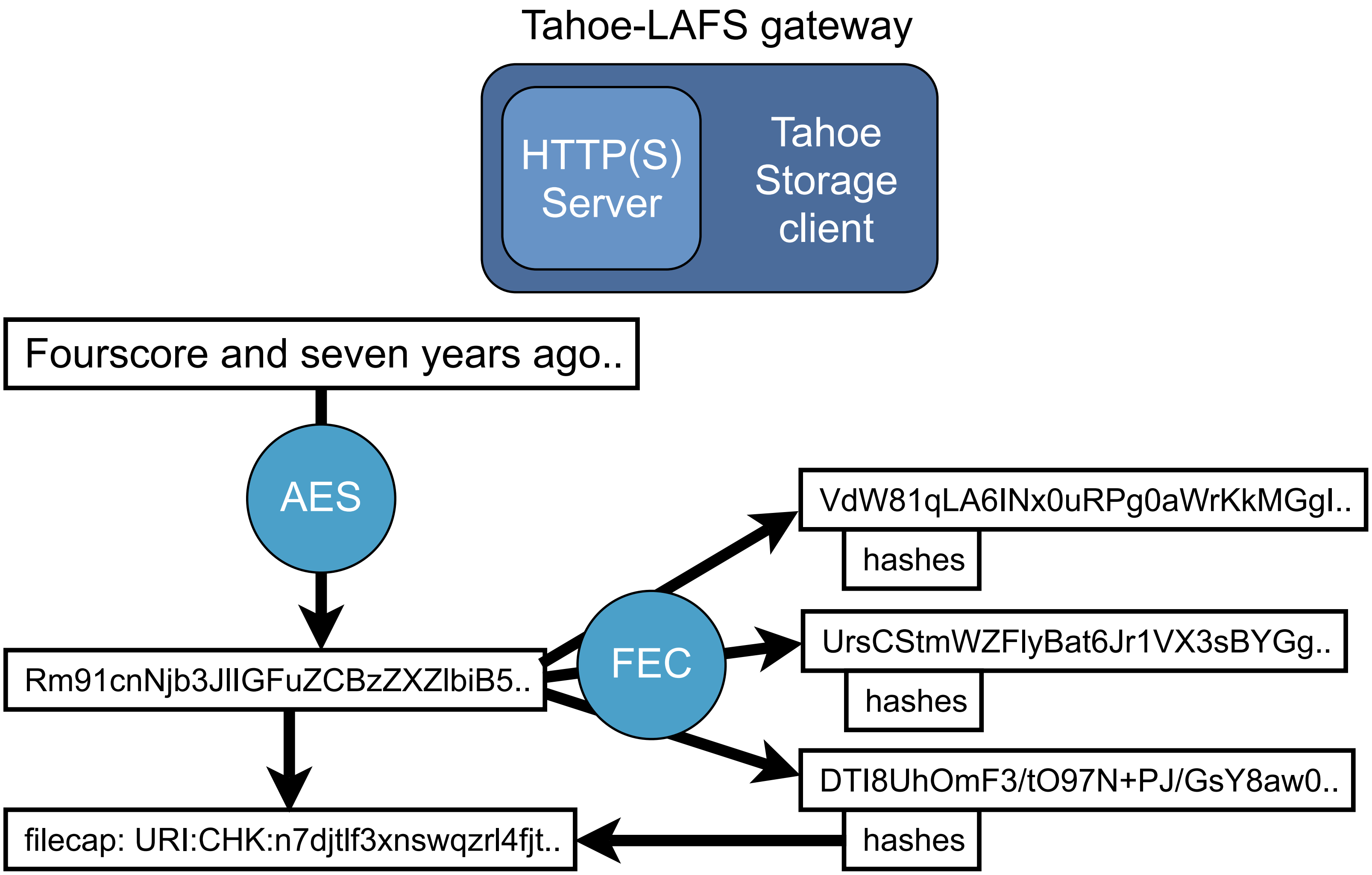


- web browser
- command-line tool
- Javascript frontends

```
PUT /uri HTTP/1.0

Fourscore and seven years ago..
```

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

File Encoding



Tahoe-LAFS gateway

HTTP(S) Server

Tahoe Storage client

Fourscore and seven years ago..

AES

Rm91cnNjb3JlIGFuZCBzZXZlbiB5..

FEC

VdW81qLA6INx0uRPg0aWrKkMGgI..

hashes

UrsCStmWZFlyBat6Jr1VX3sBYGg..

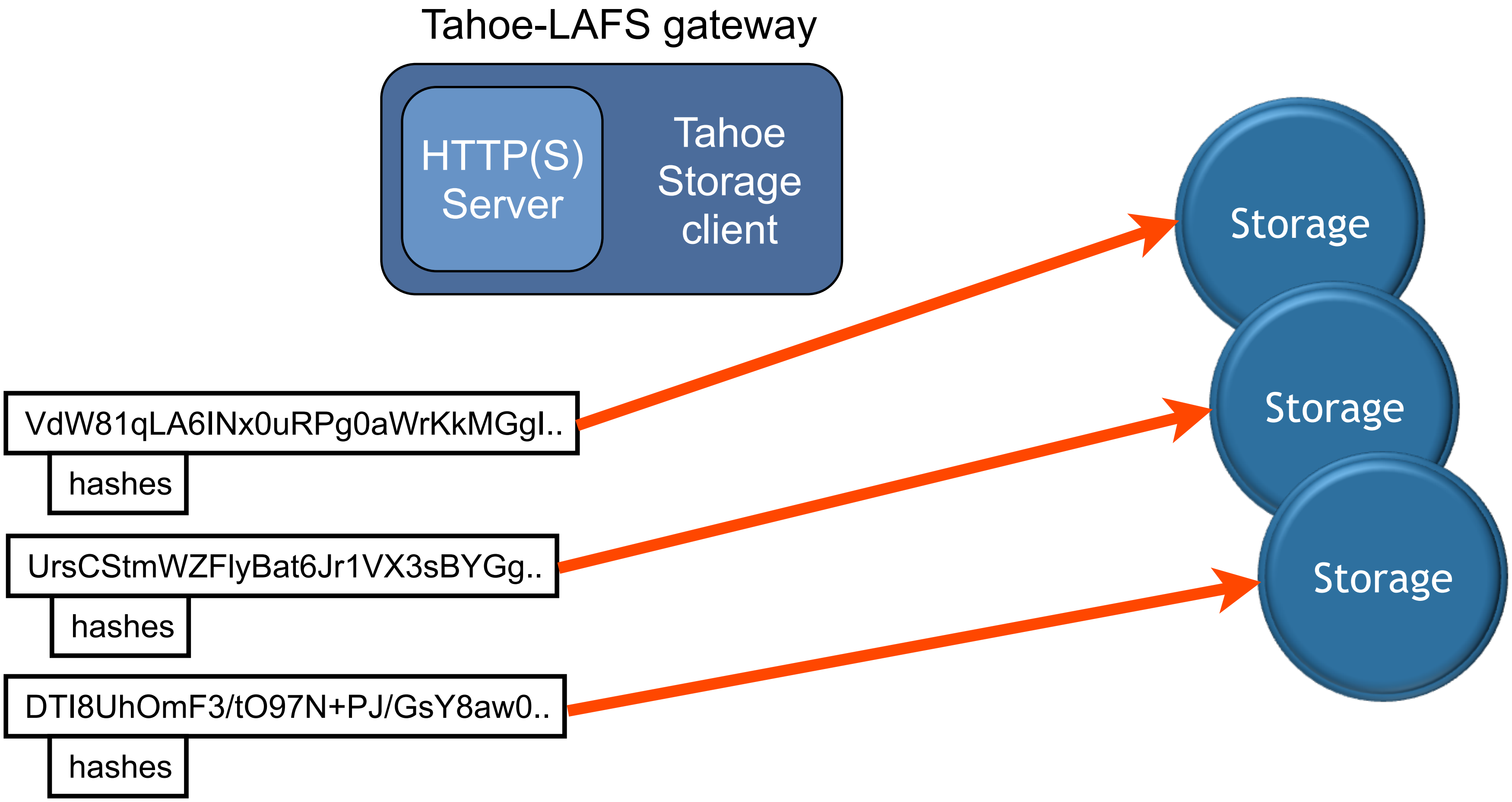
hashes

DTI8UhOmF3/tO97N+PJ/GsY8aw0..

hashes

filecap: URI:CHK:n7djtlf3xnswqzrl4fjt..

Share Upload



Tahoe-LAFS gateway

HTTP(S) Server

Tahoe Storage client

VdW81qLA6INx0uRPg0aWrKkMGgI..

hashes

UrsCStmWZFlyBat6Jr1VX3sBYGg..

hashes

DTI8UhOmF3/tO97N+PJ/GsY8aw0..

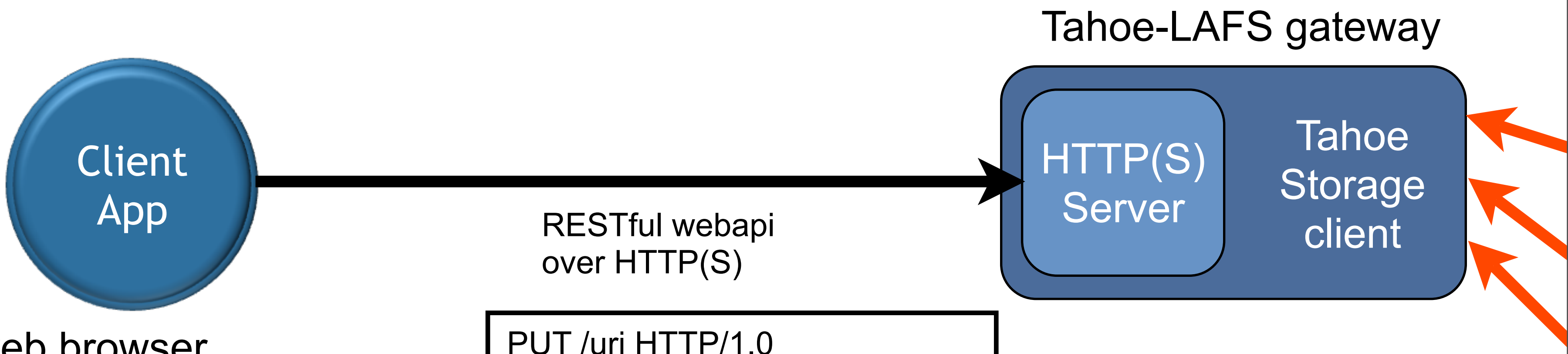
hashes

Storage

Storage

Storage

CLI returns filecap



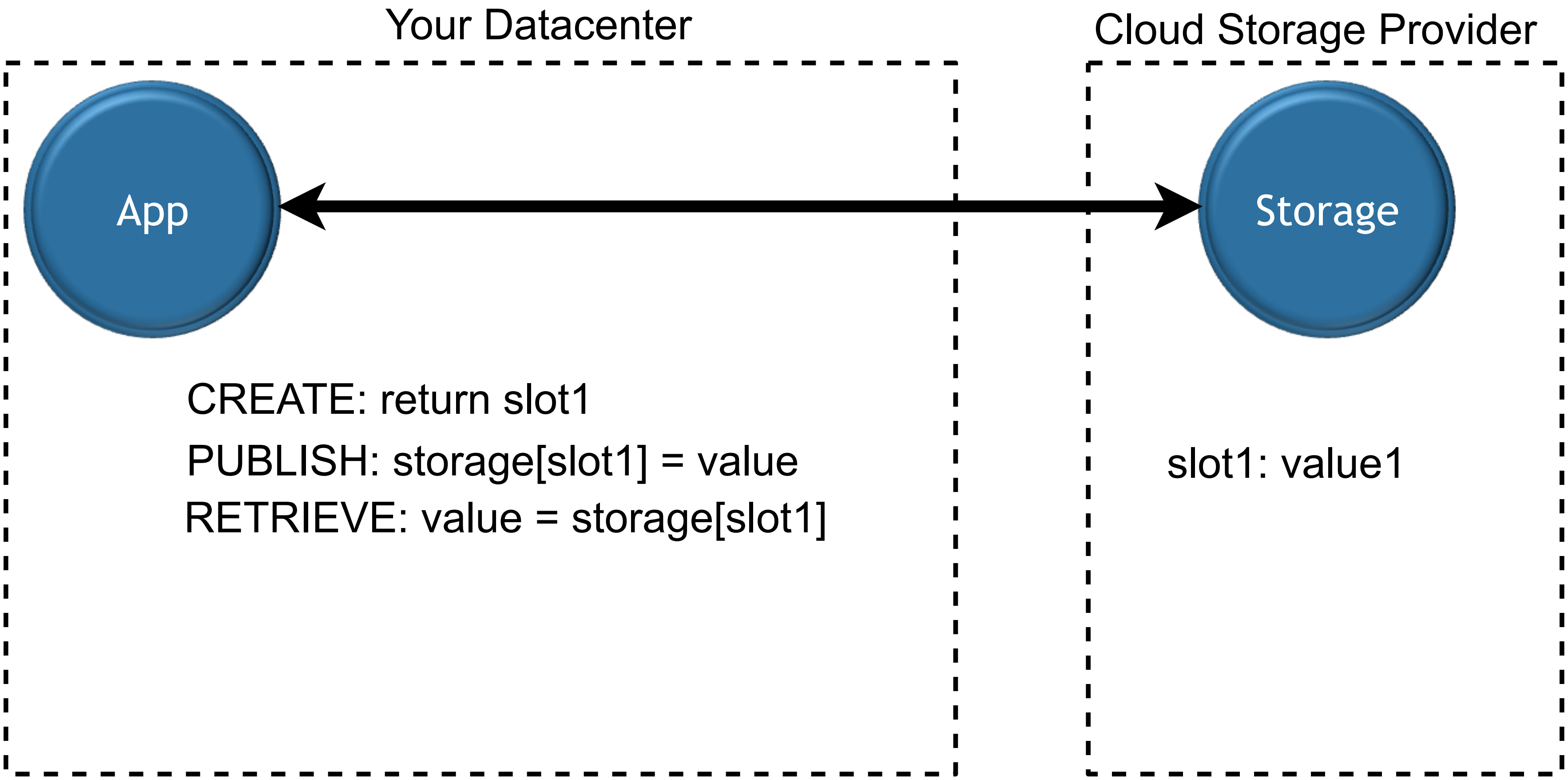
- web browser
- command-line tool
- Javascript frontends

```
PUT /uri HTTP/1.0

Fourscore and seven years ago..
```

```
% tahoe put gettysburg.txt
URI:CHK:n7djt...
%
```

```
200 OK

URI:CHK:n7djtlf3xnswqzrl4fjthqejdm:ffyrg5nbmubbjyc
6vgf3a7bgqpq24p6fzpuygp2isq2mjtgyalma:3:10:7326
```

Mutable Files



CREATE: return slot1

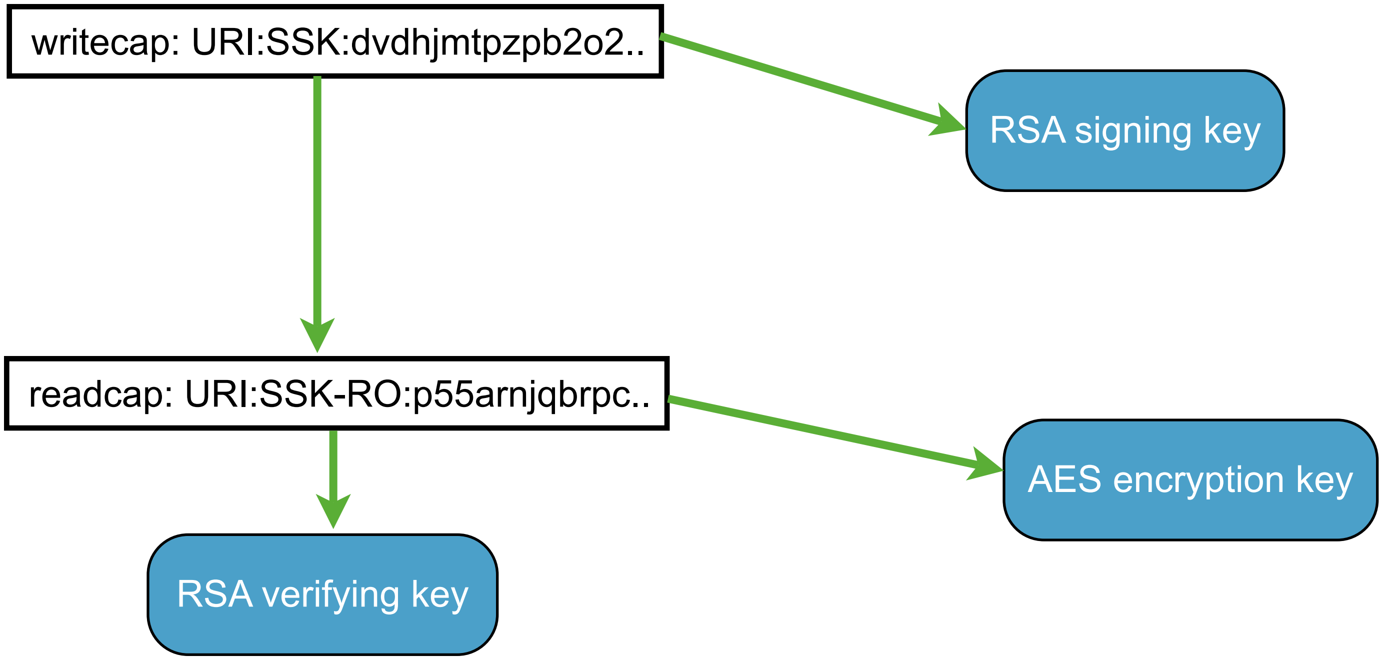
PUBLISH: storage[slot1] = value

RETRIEVE: value = storage[slot1]

slot1: value1

Mutable Filecaps

- We define two kinds of handles for mutable files
  - “writecaps” allow publishing new contents
  - “readcaps” allow retrieving existing contents
  - readcap can be derived from writecap, but not vice versa


writecap: URI:SSK:dvdhjmtpzpb2o2..

RSA signing key

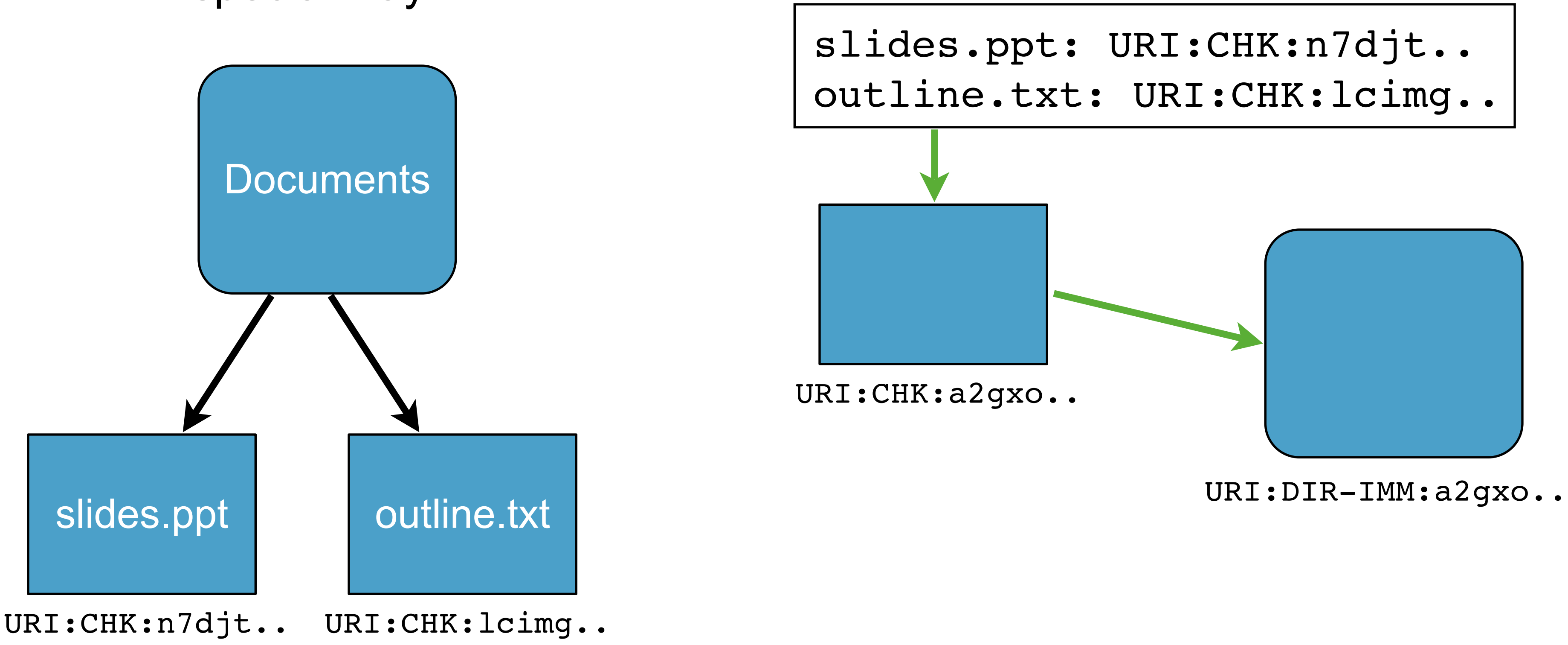
readcap: URI:SSK-RO:p55arnjqbrpc..

AES encryption key

RSA verifying key

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

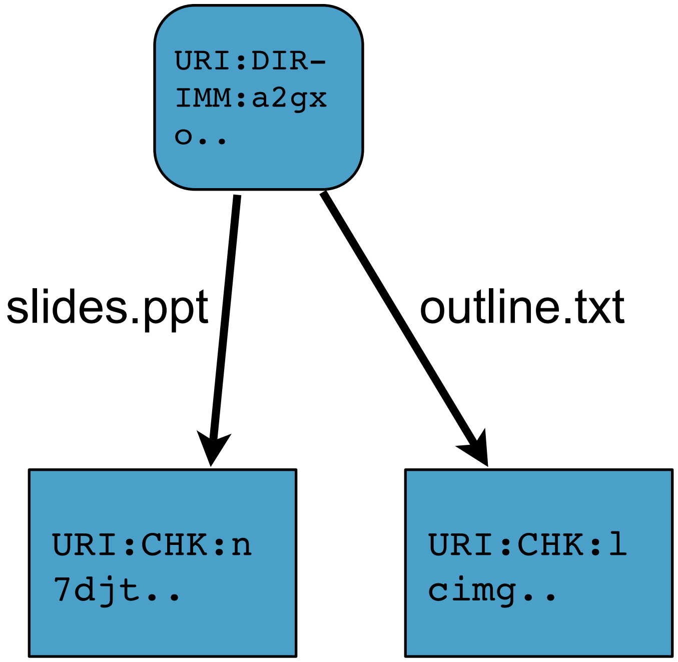


Tahoe Directories

- Directories can be stored in mutable or immutable files
  - when stored in immutable, Tahoe enforces deep-immutability
- Child caps can be readcaps or writecaps
  - 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
  - while retaining write access for themselves or others

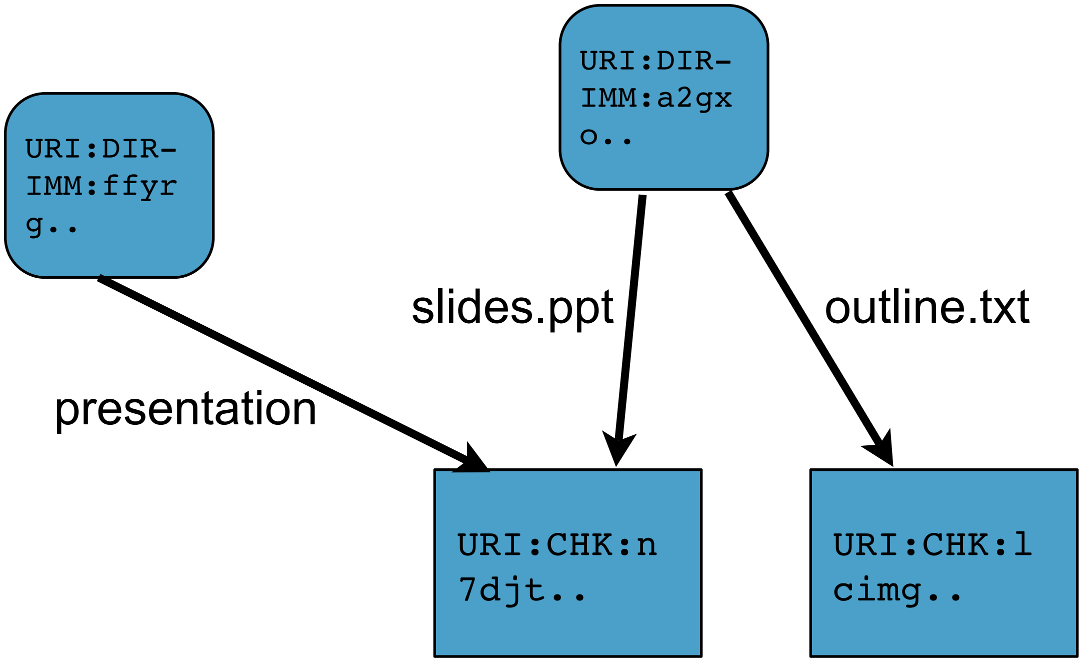
Tahoe Directory Graph

- Tahoe files and directories form a directed graph
  - names are on the edges
  - nodes are filecaps or dircaps
  - no “parent” pointers



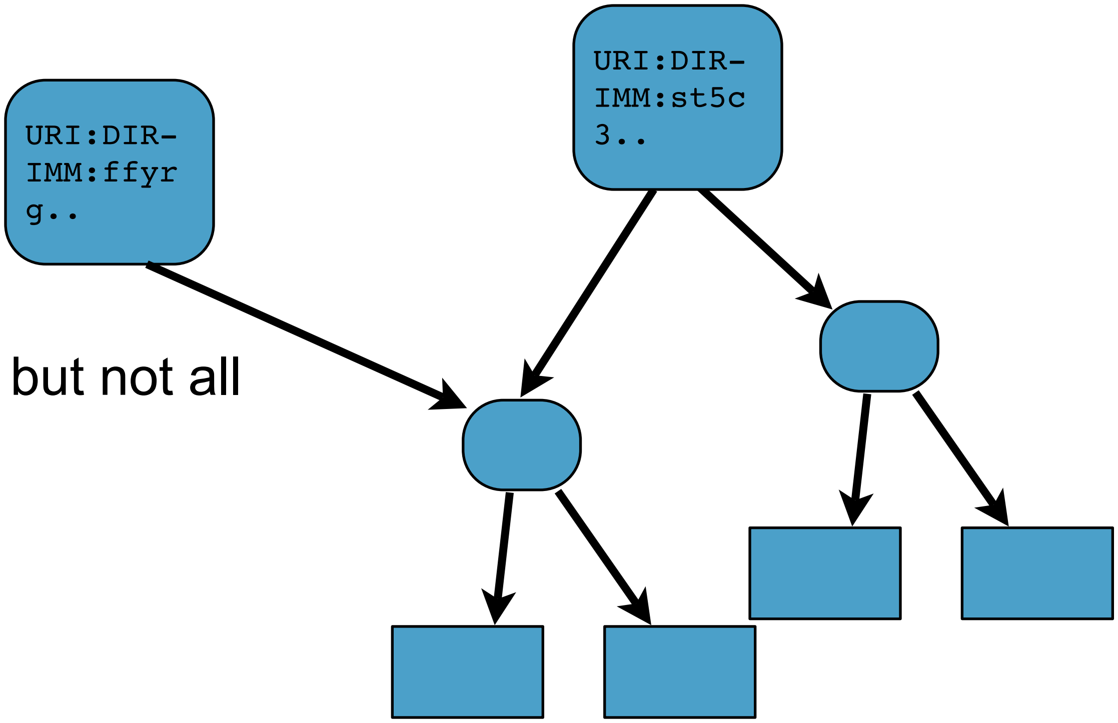
Sharing

- Files can be referenced by multiple parents



Sharing Directories

- Directories can be referenced by multiple parents
  - entire subgraphs too
  - Users can care some, but not all



Verifycaps

- All files have a “verifycap”
- 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

Ongoing Work

- Smaller filecaps, Faster mutable files
  - new formats, ECDSA, semi-private keys
- Accounting
  - tracking+controlling how much space is consumed by each user
- Garbage Collection
  - leases on shares, updated periodically, shares expire
- More frontends
  - WebDAV, Browser plugins

Demo

- On-stage Tahoe grid with several nodes
- node1: files uploaded/downloaded
- node2: shares examined, corrupted, deleted
  - files remain private, intact, available
- node1: file repair performed, new shares uploaded
- node2: replacement share examined

More Info?

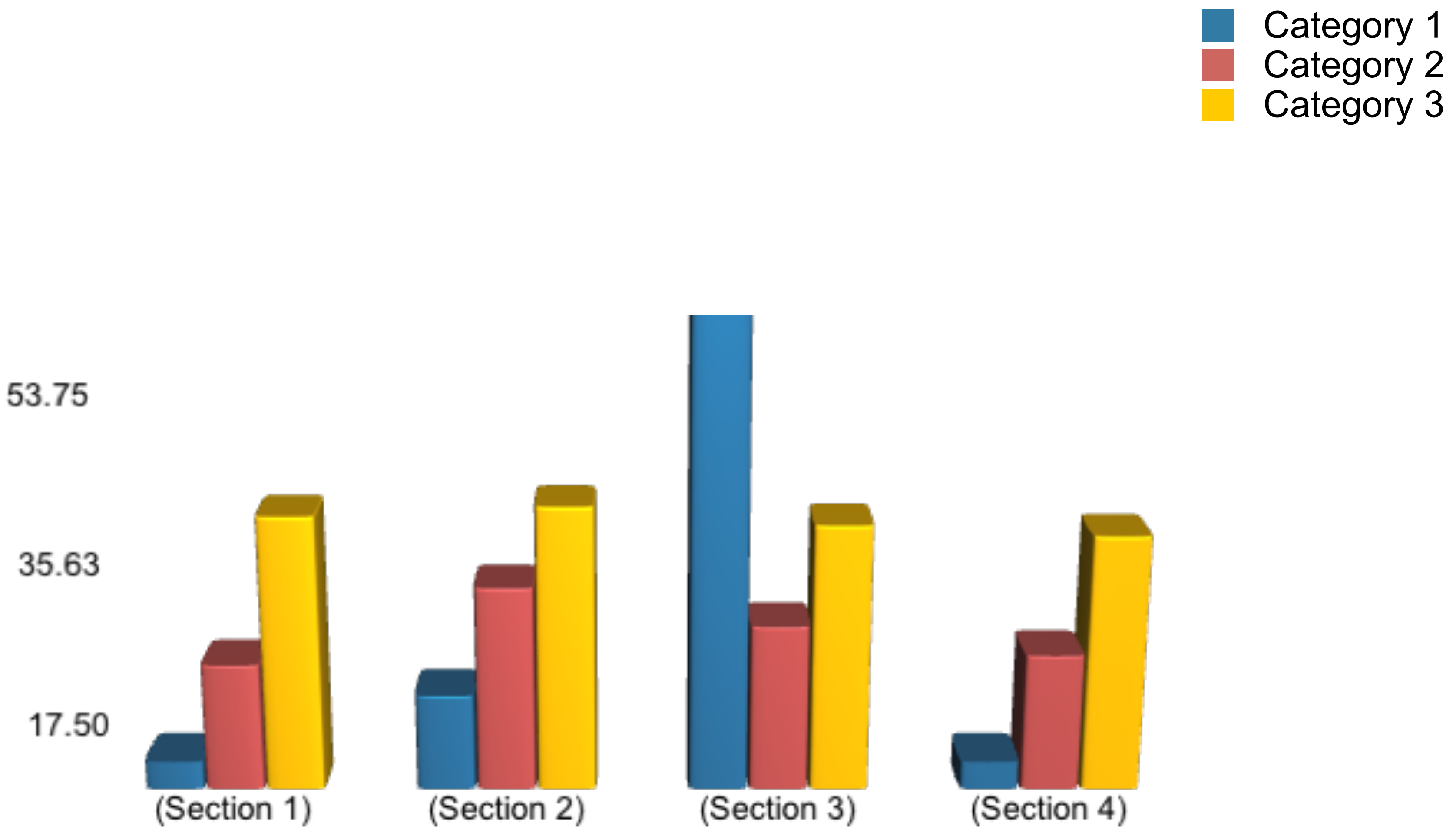
http://allmydata.org/trac/tahoe

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

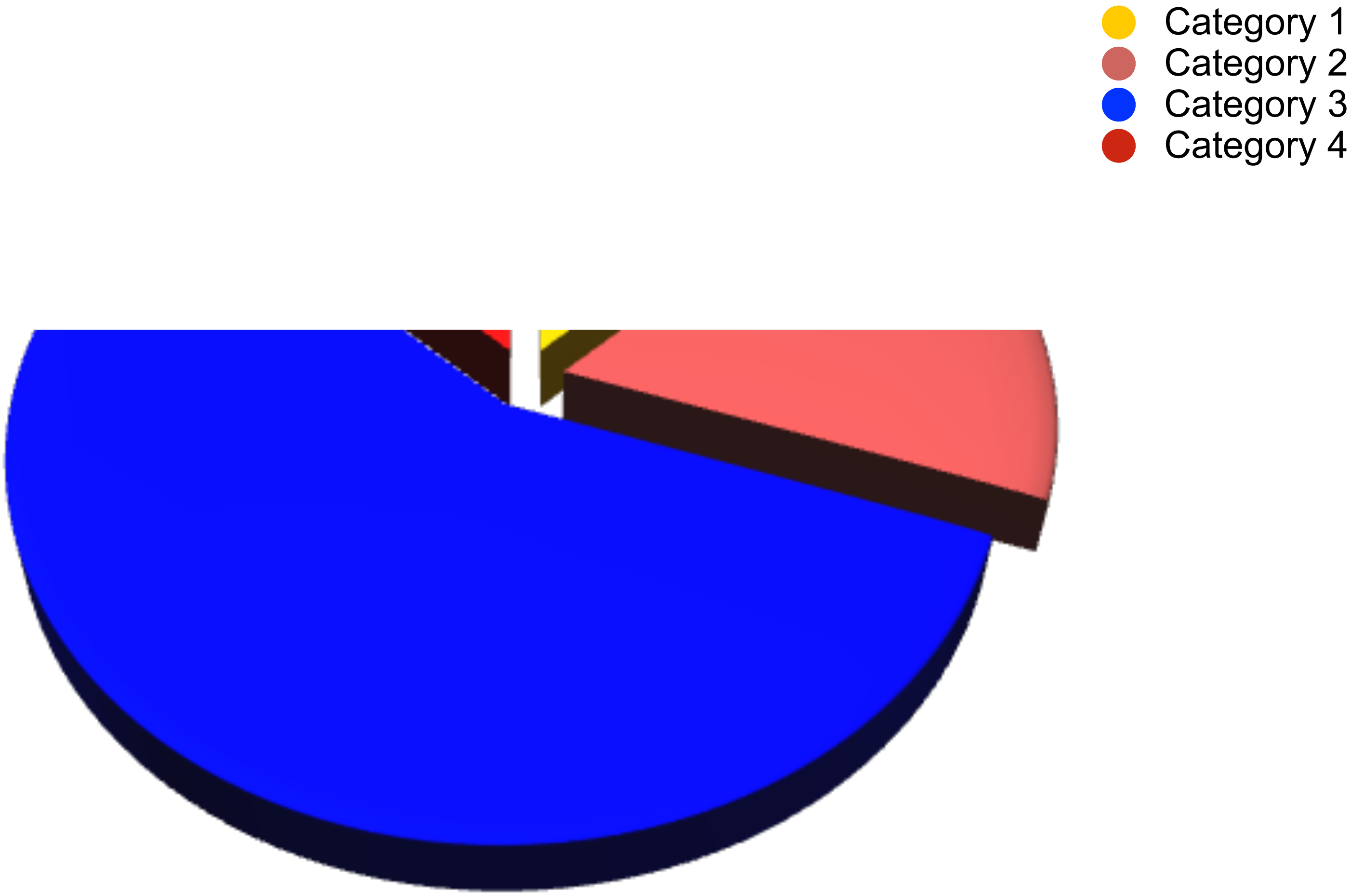
APPLY SLIDE

- Bullet point here (see slides 4 and 5 for instructions)
- Bullet point here
- Bullet point here

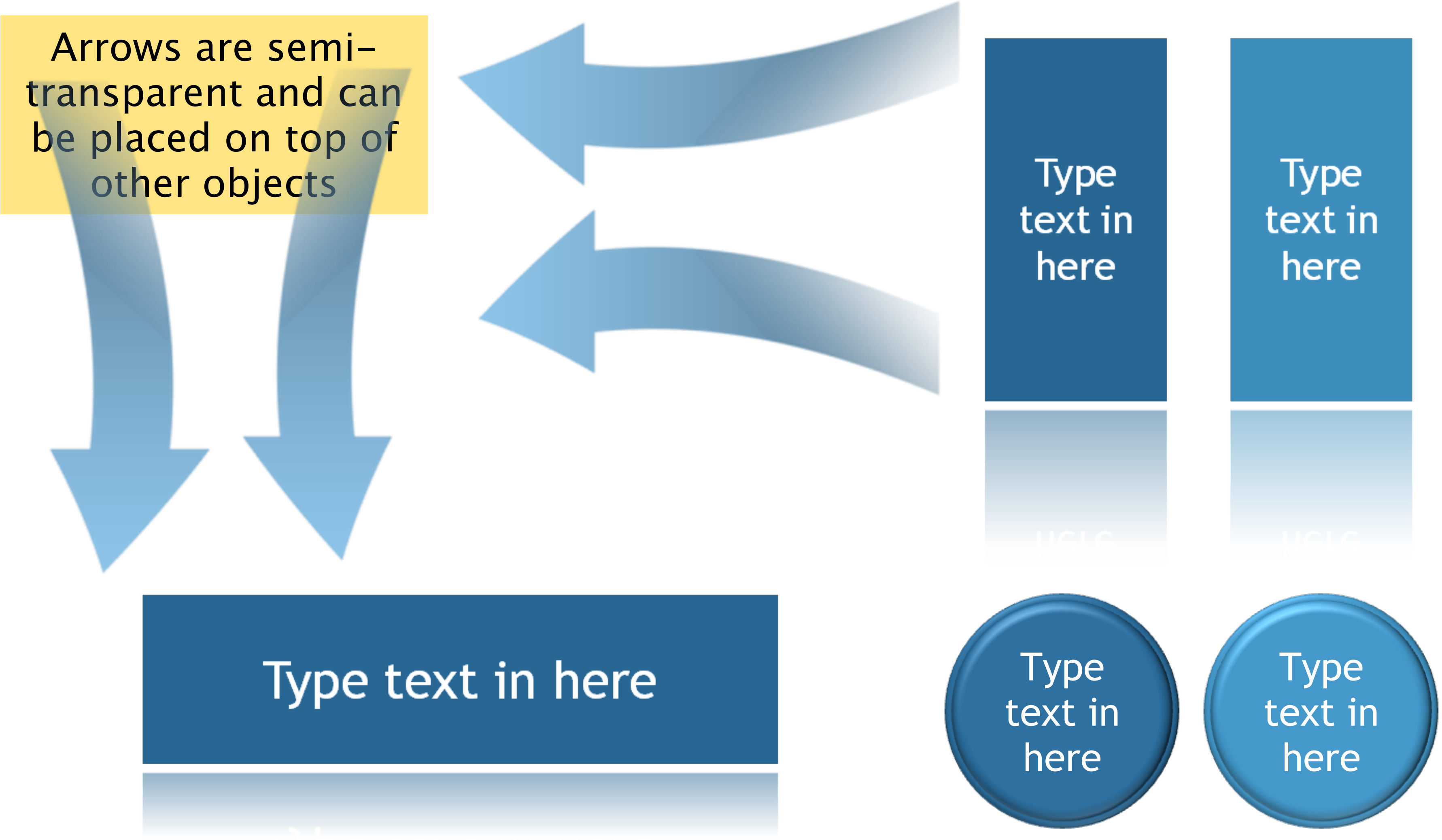
YOUR HEADLINE HERE (UPPERCASE)



YOUR HEADLINE HERE (UPPERCASE)



USEFUL ART - COPY, PASTE, AND RESIZE AS NEEDED



Arrows are semi–transparent and can be placed on top of other objects

Type text in here

Type text in here

Type text in here

Type text in here

Type text in here