Designing a System to Store Both Files and Objects

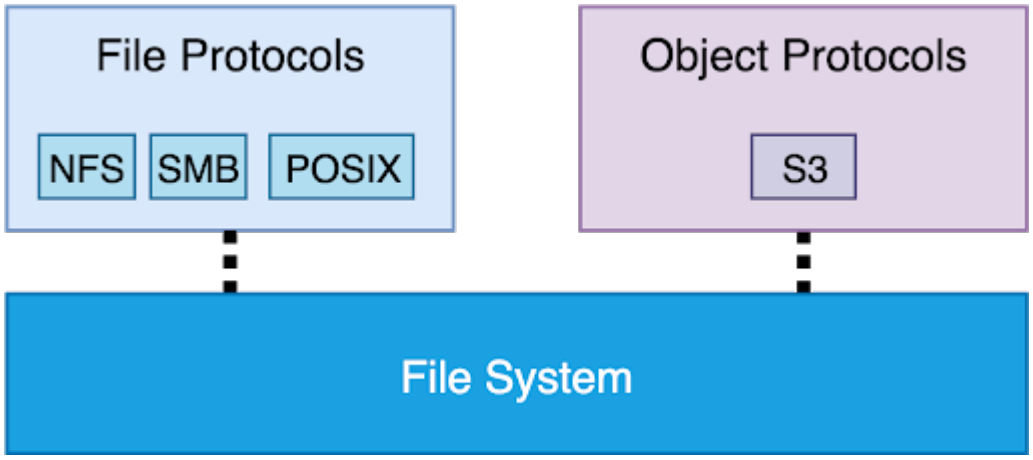
This three-part blog series is focused on a key design decision in a system that can store both files and objects: *How should a single unified design represent files, directories, objects, and buckets?*

In my previous post, I covered some background knowledge about files, objects, and buckets. In this post, we'll look at a few design options available in this space.

The options below reflect the choices of real-world storage systems.

Option 1: Object-Protocol-on-File

One option to support both file and object in one system is to build a file system and then add an object protocol as another access method:



In this design, you have a real file system that's hierarchical and transactional and features all the other file system properties that are good to have. Then, an object is represented as a file.

What does this mean in practice? Let's say that you put the object Pictures:Spring-2020:0001.JPG through the object interface. Since ultimately, the back-end store is a file system, we need to store this as a file in a directory. Arguably, the most sensible thing to do would be to configure the colon character (:) as the delimiter for the bucket. Then, the system will ensure the directory /Pictures/Spring-2020 exists and put file 0001.JPG into it.

This can work, depending on how you use the system. If the application is using the system as if it were a file system, just using the object API as the access point, this will work fine. But, if the application is actually using the bucket as an object store (i.e., a flat namespace), you'll end up with a single massive directory.

As with any approach, there'll be some pros and cons:

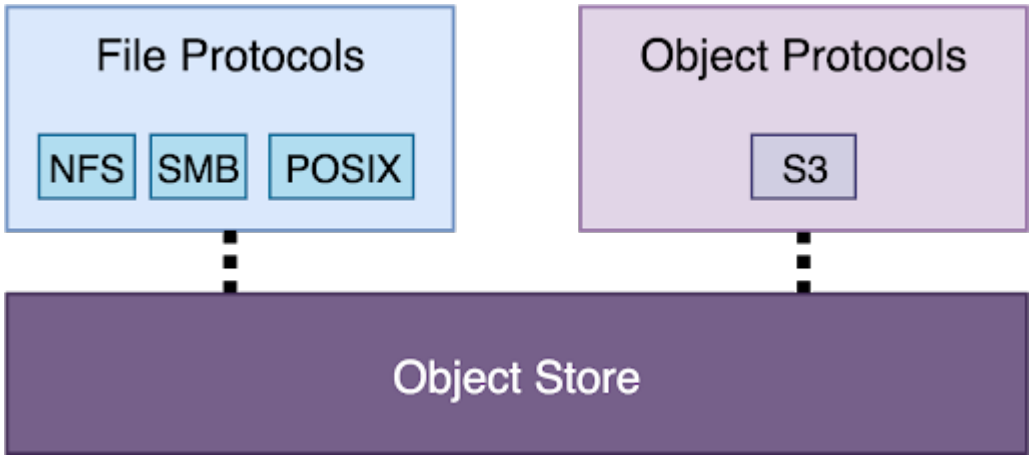
Pros	Cons
No-compromise File The architecture can support a highly compliant, robust file system. **Interoperability** The architecture naturally supports object-file cross-protocol access.	**Limited Object** This architecture probably won't lead to a highly compliant, well-performing, scalable object store. **Object Features** There will be limitations and challenges in adding object-specific features in the future because the core of the system is file-based.

Ultimately, this approach is attractive to systems that are primarily oriented towards file workloads, but want to add some object support. For example, in an environment that's mostly based on file, if there's an application that wants to read the files as objects, this approach is suitable. Also, it'll work with specific applications that have been qualified for the object store.

However, this choice is unlikely to lead to a highly compliant, scalable, and general-purpose object store. A bucket requires a massive, flat, ordered namespace, and that's not something a file system provides. Instead, the file system incurs the cost to enforce transactionality and other properties that are irrelevant to object storage. Object storage also depends on per-object versioning and replication, concepts that don't map well to file systems. So, this architecture can serve some object use cases, but ultimately it won't be a first-class object store.

Option 2: File-Protocol-on-Object

Now let's look at the completely opposite option: building the system as primarily an object store. Each file will be represented as one object in the underlying object store:



This design is attractive if you want to build a system that's primarily focused on object storage. The file semantics are going to be very limited because the rich, transactional properties of file systems simply cannot be expressed in the world of objects and buckets. For example, there's no way to implement an operation like renaming a directory with faithful semantics.

Let's look at the pros and cons of this approach:

Pros	Cons
No-compromise Object The architecture can support a highly compliant, robust object store. **Interoperability** It's pretty easy to support object-file cross-protocol access in this model.	**Very Limited File** The file semantics will be very limited. Most of the issues will be basically unfixable.

In practice, this choice makes sense if you're trying to build a system that's primarily an object store. The file protocol support will enable users to browse and download objects as files. But, it won't be usable as a storage back end for applications: For example, something like a build simply won't work.

Example

If you're curious about the types of problems encountered, consider this bucket:

```
projects/hello_world/main.cpp
projects/hello_world/hello.cpp
projects/hello_world/include/hello.cpp
```

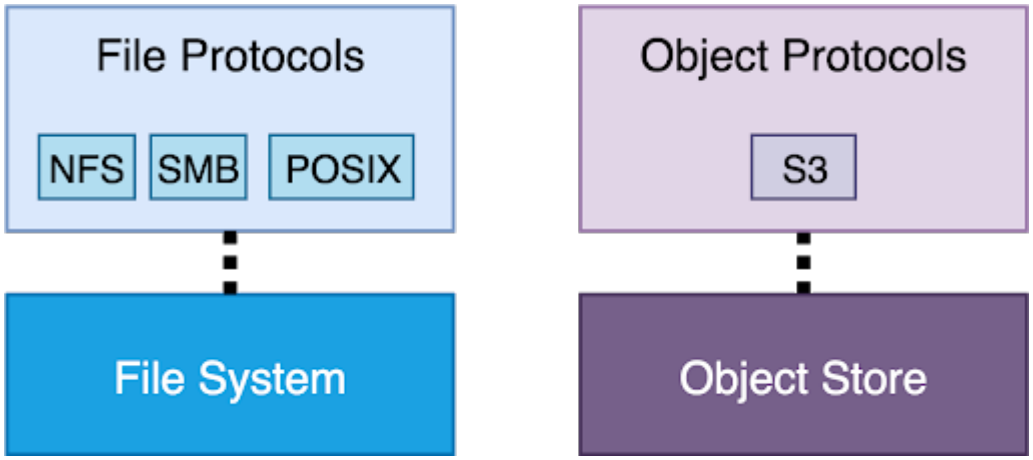
Interpreting the bucket as a file system, let's say that we want to rename the directory hello_world to hello. Well, the only way to express this operation is to rename all three objects to reflect the new name of the directory. However, this implementation would break important file system properties, such as the following:

- The operation needs to be atomic.
- Modification time on the parent directory needs to be adjusted correctly.
- If the target directory exists, the rename must fail exactly if the target is non-empty.

There are many of these types of subtle properties. And applications depend on these properties, often in subtle and complex ways.

Option 3: File and Object

Here's another option that's based on two separate stores: a file system and an object store:

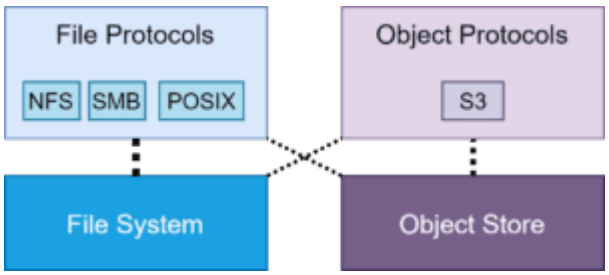


In this model, there are two completely separate core back-end stores: a file system and an object store.

Pros	Cons
No-compromise File The architecture can support a highly compliant, robust object store. **No-compromise Object** The architecture can support a highly compliant, robust object store.	**Two Back-end Stores** The system needs to implement separate file and object stores. Two stores are more work than one. **Interop Is Not "Free"** The architecture doesn't immediately provide cross-protocol access. That would be extra work to add.

In this architecture, we get a no-compromise file store and a no-compromise object store. You create file systems that can be accessed via file protocols, and buckets that can be accessed via bucket protocols. But, the file and object namespaces are separate, so you can't read the files over object protocols and vice versa.

Of course, nothing prevents you from adding cross-protocol access as well. You can also implement file protocols for the object store and object protocols for the file store. In the end, the architecture can evolve to this: In this architecture, we get a no-compromise file store and a no-compromise object store. You create file systems that can be accessed via file protocols, and buckets that can be accessed via bucket protocols. But, the file and object namespaces are separate, so you can't read the files over object protocols and vice versa.



In this model, you choose whether the data set should have file semantics or object semantics, but they can ultimately be accessed by both protocols.

In the third and final part of this series, we'll describe the architectural choice we made for FlashBlade.

For a broader context, see the post Unified Fast File and Object: A New Category of Storage.