Abstract

Colorhythm, LLC has expressed interest in supporting development of a "cloud-friendly" version of ResourceSpace. In order to do this, steps must be taken to abstract the permanent storage of resource files from the core ResourceSpace application, and to support direct delivery of files to users from a remote storage location, such as Amazon's Simple Storage Service (S3).

Part of the goal of this project will be to retain full backward compatibility with the existing local filestore concept, and to make the new capabilities largely invisible to end users. However, some changes to the core system architecture will be necessary, and future developers may need to be aware of the possibility of remote storage when making changes to code.

This document will attempt to lay out an overall strategy for the changes being considered, and will solicit comments and feedback from the broader ResourceSpace community on the proposed changes.

Making ResourceSpace "Cloud Friendly"

Currently ResourceSpace is designed primarily to run on a single server with locally-attached storage. In the most common configuration, the MySQL database, web server, and resource filestore are located on a same physical machine. In some situations, files may be stored using a network-attached storage device, or a fileshare accessed over a LAN via a protocol like NFS or SMB. But even in these cases, the system requires that the filestore appear as a directly-accessible block-level storage device.

The reliance on local file storage limits the flexibility of the current architecture. Creating a more flexible model could encourage development of novel approaches to ResourceSpace storage, such as geographically distributed installations, load balanced clusters, etc. This document will focus primarily on one such application: the implementation of ResourceSpace on the Amazon Web Services (AWS) platform, using Amazon Simple Storage Service (S3) for remote object-based storage of resources and previews.

Amazon AWS, and, more broadly, cloud computing, focuses on breaking computing tasks into their functional components, provisioning resources as needed to meet those demands, and then often decommissioning resources when they are no longer needed. For example, in the AWS architecture, virtual servers (EC2 instances) are considered to be somewhat ephemeral. Servers can come and go as needed, with new instances being added or removed in response to demand. Under this model, you might have one MySQL database (using Amazon RDS) serving 5 load balanced EC2 instances all running ResourceSpace, doing transcoding, etc. When a big batch

of video is ingested and needs transcoding, you can spin up a few more servers, do the work quickly, and then shut them down again to save cost and computing resources. Meanwhile, the important files and database data are stored "off instance" using cloud-based storage tools like RDS and S3.

Amazon S3

While a number of the AWS services provide ways to store data, the preferred method for persistent storage of large digital objects is Amazon Simple Storage Service (S3). S3 is a web-service based system that stores and retrieves objects in as monolithic units. A client application can store a file in S3, and the file will receive a unique URL. It can then be retrieved at this URL at a later date. Unlike some other AWS components like EBS and EC2, which are designed to be ephemeral, S3 is considered a high-availability storage platform. The system is designed for 99.999999999% durability and 99.99% reliability over the course of a year. The system can also provide robust versioning of files, and can interact with content distribution networks, access control subsystems, and streaming media services.

A single S3 storage pool is known as a "bucket." S3 does not natively index the content of these buckets, so an application utilizing S3 storage should maintain an inventory of objects stored in order to retrieve them efficiently. Furthermore, S3 provides no mechanism for incrementally modifying files stored in the service. So if an application wishes to modify the content of a stored object, it must download it, make the change, and re-upload it.

The fact that objects stored in S3 are directly accessible via HTTP changes the way the stored files can be used for web applications. In the current model, for example, the ResourceSpace server reads each thumbnail off the local disk and sends it to each client each time the resource is viewed. However, if the thumbnails were accessible in the S3 cloud, the server could simply provide the URL to the client, and the thumbnails would be downloaded directly from S3. This would reduce the amount of local processor and network bandwidth used by the server, as well as the local storage requirements for the application.

Currently, an experimental S3sync plugin is available for ResourceSpace systems running on Unix-based platforms. This add-on facilitates syncing the filestore to a remote S3 bucket, which can be useful for backup. But ResourceSpace still expects to have a complete copy of the filestore on locally-attached disks. This precludes taking full advantage of cloud storage. It may also make it more difficult to scale the system to the hundreds of terabytes of storage required for large scale video archives or other large scale data repositories.

Can ResourceSpace Run on AWS Now?

Yes, but….

It is possible to run ResourceSpace on an Amazon EC2 instance, but many of the advantages of the cloud (scalability, performace, etc.) are lost. Since Amazon does not guarantee the long term durability of EC2 or EBS storage, care must be taken to frequently snapshot data stored on these platforms to S3 images, which adds complexity and cost. Furthermore, data stored in these services lose the availability, scalability, and other benefits of the S3 storage platform.

There are hacks that can facilitate storage of block-based data in S3 (such as S3fs), but these also involve tradeoffs – for example, as of early 2011 files stored by S3FS cannot be browsed by common S3 tools like S3Fox, and files imported via Amazon's AWS Import/Export service cannot be seen by S3FS. It is also more difficult to take full advantage of add-on services like S3/Cloudfront flash streaming server in this type of setup.

The applications that work best with AWS (and with other cloud-based services) are those that are designed with the unique considerations of cloud-based installations in mind. The primary goal of this project is to make ResourceSpace work well with S3, and with other object-based remote storage architectures.

Conceptual Requirements for Cloud Friendly ResourceSpace

1. When creating new files, upload them to remote storage (and optionally delete them from local storage.) This may be done in realtime or via a behind-the-scenes process.
2. Maintain a registry of all available files associated with a resource (including previews) and the location(s) in which they are stored.
3. When determining if a file is available, check this registry in addition to the local disk.
4. If a file is available remotely and does not need to be modified prior to download, serve it directly from the remote location, reducing load on the local server.
5. If the file must be modified, download a local copy, make the change, and then flag for re-upload to the remote storage service, recording this activity in the registry so that it's clear where the most recently modified copy is.

In place of the current reliance on direct access to all resources and files via the local filesystem, we envision implementing a system of "storage plugins." These plugins will abstract the process of storing and retrieving files from the core application, making it easier to make use of remote storage services, implement clustering, store deposited files in established in existing institutional digital repositories like Fedora Commons, support geographically distributed ResourceSpace servers, etc.

Storage plugins will be required to implement certain functions. These including storing files, retrieving files, updating previously stored files, providing a URL to a file, etc.

The goal will be to retain 100% backward compatible with the existing storage scheme, so that users who continue to use local block-based storage will notice no differences. However, some architectural changes are likely to be necessary to make it work. Most notably, developers will no longer be able to rely on PHP filesystem functions like "file_exists" to determine the availability of previews, etc. Instead, they will need to call ResourceSpace functions that are aware of the various remote storage options.

Also, when they access a file, developers will need to decide whether they actually need the file on the local web server, or can simply send it directly to the user from remote storage. And for portions of the system that modify or create files, care will need to be taken to update the master file inventory after performing these operations.

Development focus

While the initial buildout being funded by Colorhythm, LLC will focus on enabling ResourceSpace to run effectively on the Amazon Web Services platform, the goal will be to build the new architecture in a way that will allow the system to support other storage technologies in the future – clusters, private storage clouds, etc.

While more robust support for cloud computing could help support very large ResourceSpace installations, it could also enable new capabilities for ResourceSpace users of all levels. In theory, for example, it would become far easier for someone to run a minimal ResourceSpace installation on a low cost web hosting account, while still having virtually unlimited storage capability via Amazon S3 or another storage provider.

Implementation considerations

1. Storage plugins will be developed for Amazon S3, while retaining full support for local storage.
2. Each use of file_exists or other PHP file functions must be substituted for a new, more nuanced functions provided by the storage plugins. These will consider the availability of files on remote storage. (approx. 75 files use file_exists function in system and core plugins)
3. Use of the get_resource_path function should be evaluated to determine if a remote URL could be substituted for the local path or URL. It may be possible to implement some of these changes seamlessly within the function. This function is us used in 57 files in the system and core plugins.
4. Each use of an external helper app (such as imagemagick, exiftool, etc.) must be evaluated to determine if it requires local presence of the files, and, if so, must first ask the storage plugin to "localize" the file. (approx . 35 files contain the shell_exec function in system and core plugins; most of these likely represent calls to helper apps that may expect files to be present on the local filesystem)

5. Each operation that creates a new file or changes the permanently-stored version of the file (such as transforming original) must create or flag an entry in the new file registry for upload to remote storage. In most cases, this will be taken care of by core system functions, but in some situations the developer may need to explicitly flag that a file has changed.

Efforts will be made to ensure that the new architecture does not break existing plugin code for those who do not choose to take advantage of remote storage capabilities. However, as with any architectural change, developers and users will be advised to test the update carefully before deploying it.

Project Status

This project is currently in the early planning stages. Comments are being sought from the broader ResourceSpace community, with the goal of developing a more complete project plan in the near future.

Please direct questions or comments to the ResourceSpace Google group, or to David Dwiggins at david [at] dwiggins [dot] net.