Contents

Revision History

Initially Published: April 17, 2012

Revision 1: April 30, 2012
- Log History considerations

Revision 2: July 20, 2012
- Process Construction considerations
- Overall Goal added to Overview
- Benefits section added
- 'Compact' considerations for the fGDB
- Updated ALF and ALF-Lite Diagrams from v1.0 to v1.1

Revision 3: October 8, 2012
- Document altered to use 'Revision' numbering
- Included comments on controlling Spatial Domain
- Technique for loading large data using eGDB

Revision 4: July 15, 2014
- Updated File and Enterprise Geodatabase references to fGDB and eGDB respectively
- Spatial Index considerations and important note regarding Spatial Extent
- Time-Enabled content considerations
- Performance considerations
- Leveraging Multiprocessing
- Leveraging Feature Services

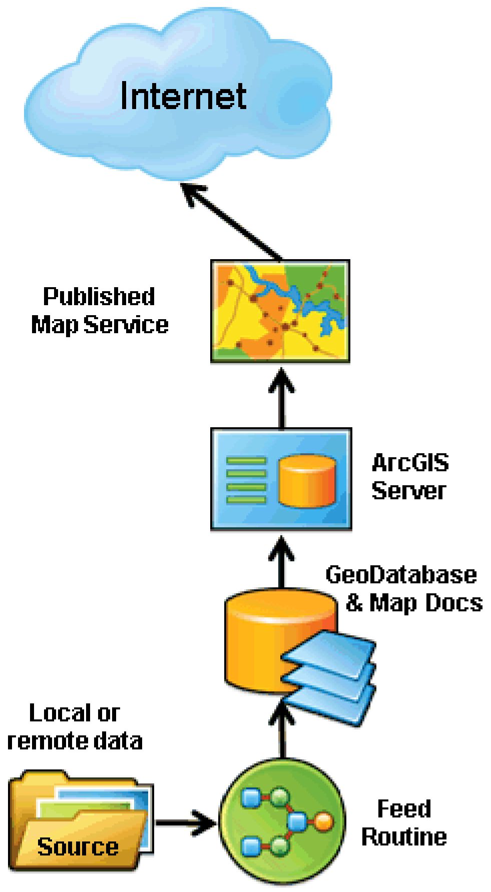
Overview

The intent of this document is to cover the detailed aspects and architecture of the available Aggregated Live Feed methodologies. The general guidelines section includes insight on the critical discoveries we've learn along the way.

This is a living document, revised as methods and procedures evolve.

The overall goal of an Aggregated Live Feed routine, simply put, is to produce a consumable data source that can be accessed and shared while being refreshed by an automated process.

The resulting data can be consumed by a single Desktop GIS user, or accessed by many users over the internet / intranet by leveraging ArcGIS Server to create a web accessible Map Service.



ALF Overview, v1.0

Benefits

Internet accessible data sources are a wonderful and convenient way to add content and value to your finished Mapping and Analysis work. If only a handful of your users leverage Internet content to complete or enhance their work, then the demands on your Internet infrastructure are fairly light and manageable. But what happens if more than a handful of your users do this? Say, hundreds or thousands are accessing remote services and content? The demands on your Internet resources could be stretched to their limit. Not to mention the potential delays that users could experience.

Aggregated Live Feeds can be leveraged to lighten the demand by brining common data sources internal to your organization, allowing your users to access faster, more reliable **Intra**net hosted services rather than external resources.

What happens if an Internet data source becomes unavailable, sources that your users rely on to complete critical tasks? Will productivity come to a standstill until the data source returns?

By retaining a local copy of the most recent content, Aggregated Live Feeds remain available, giving your users the ability to stay productive during an outage.

Aggregated Live Feeds update data automatically, no need to stop and start a service or connection just to refresh the data! Simplifies data maintenance and logistics, improves service up-time and availability.

History

Summer 2007

- The initial ***Aggregated Live Feed*** methodology was conceived and implemented by Esri's corporate Technical Marketing group in order to satisfy the need for relevant, timely data usable in product demonstrations. This methodology leverages the ArcSDE eGDB technology to host the data using a collection of Windows command-line batch scripts and GNU-based components that download, process, and import or update the data.

Fall 2011

- Given the apparent design limitations of the batch architecture, the technical understanding required for all components involved, and the eGDB resources required to successfully deploy the initial method, has lead us to develop a simpler way. Experimenting with the fGDB and the ArcGIS Geoprocessing tools available to Python, we came up with a way to manage and publish data using a simpler technique we call the ***Aggregated Live Feed–Lite*** methodology (**ALF-Lite** for short). This methodology leverages Python to handle the operational tasks, ArcGIS ArcPy to handle the GIS Geoprocessing tasks, and the lightweight fGDB to manage the data. Not requiring third-party components or the eGDB should aid audience appeal. The end-result is a fGDB that can be updated and maintained by an automated process, while Desktop and Server clients are busy viewing that same data.

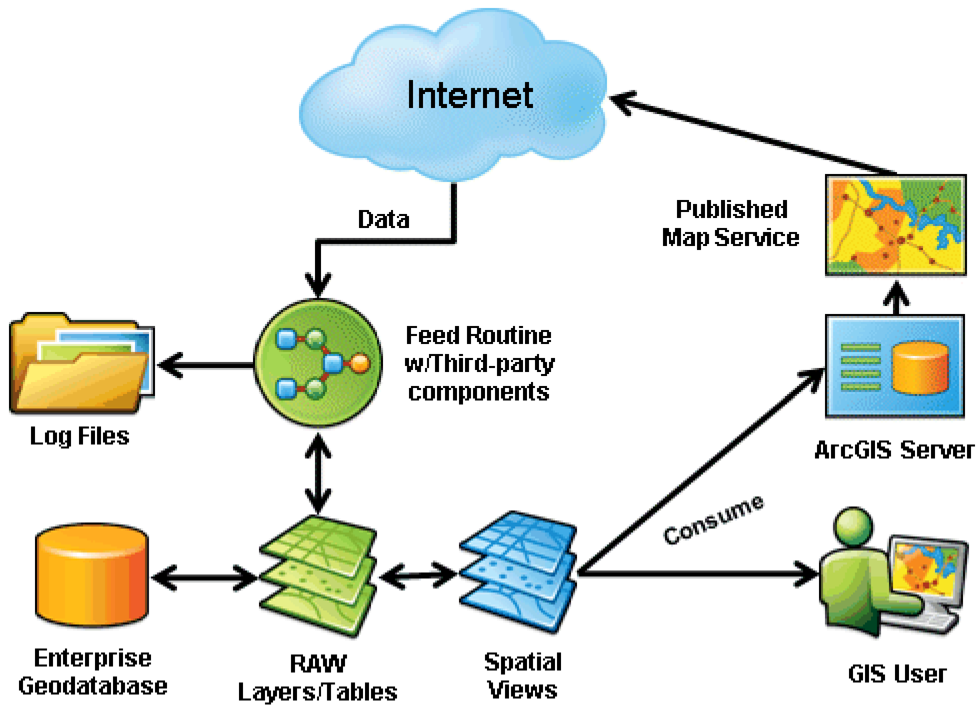
Architecture

Aggregated Live Feed methodology

- Using a command-line Web download component like Wget, data is downloaded from the source, processed using third-party components based on the data type or the type of processing require, and finally imported into the eGDB using native or custom ArcSDE command-line tools. See the 'ALF Diagram' below.

- In this design, the processed data will replace or is used to update the existing RAW data Layers or Tables.

- Spatial Views are then used to allow users access to the data without limiting or interfering with the update process. This enables the feed routine to quickly update the underlying content without restriction. Database operations like 'Truncate' and rebuilding indexes cannot be performed if another ArcGIS user is locking the Featureclass.



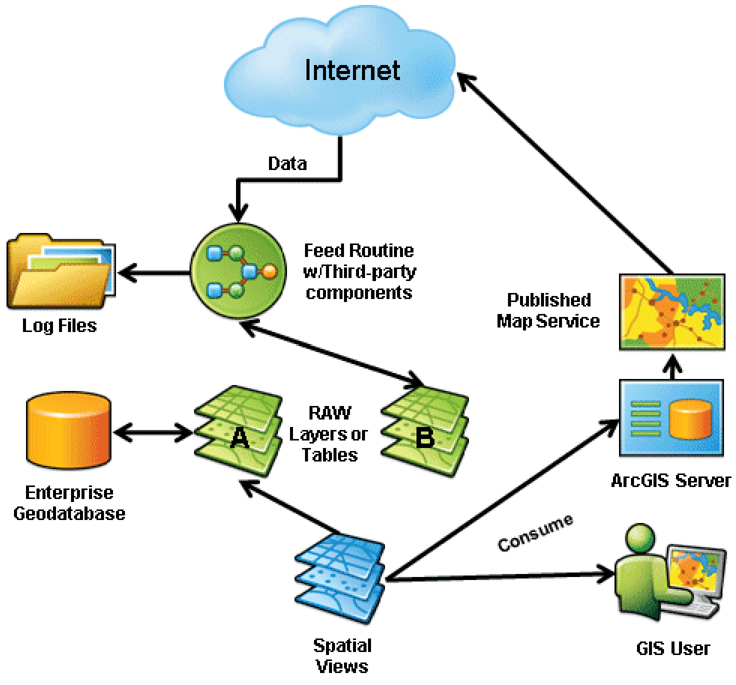
ALF Diagram, v1.1

- When dealing with large content, those likely to take longer than a second or so to update, you can employ multiple 'RAW' Layers ('A' and 'B') and a single Spatial View. The View provides access to one or the other 'RAW' Layer while the load routine Truncates and loads the other. When the load completes, the routine simply updates the View (using SQL), pointing the view to the Layer that was just loading. Depending on your Database, the SQL for this would look something like the following:

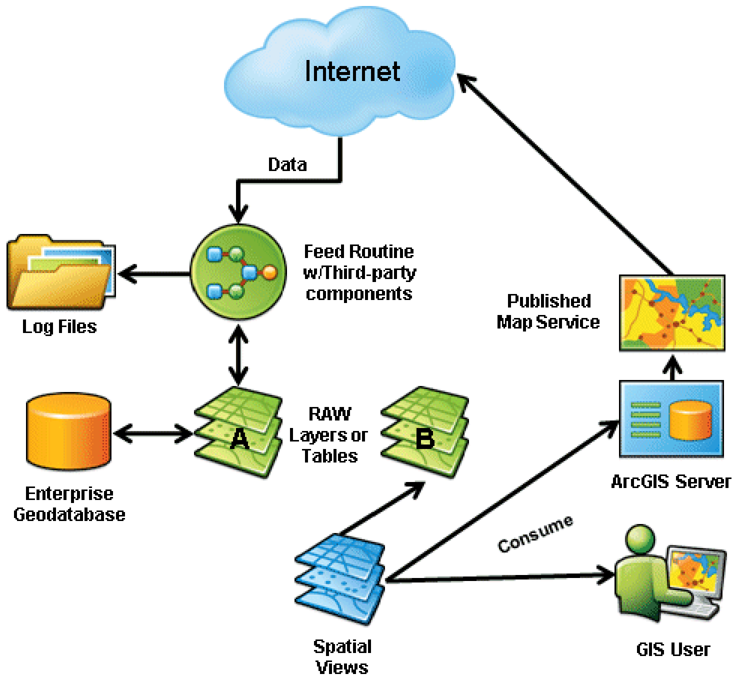
- ‘Create or Replace VIEW <mySpatialView> as Select * from <myRawLayer_A>’

  Or:

- ‘Create VIEW <mySpatialView> as Select * from <myRawLayer_B>’



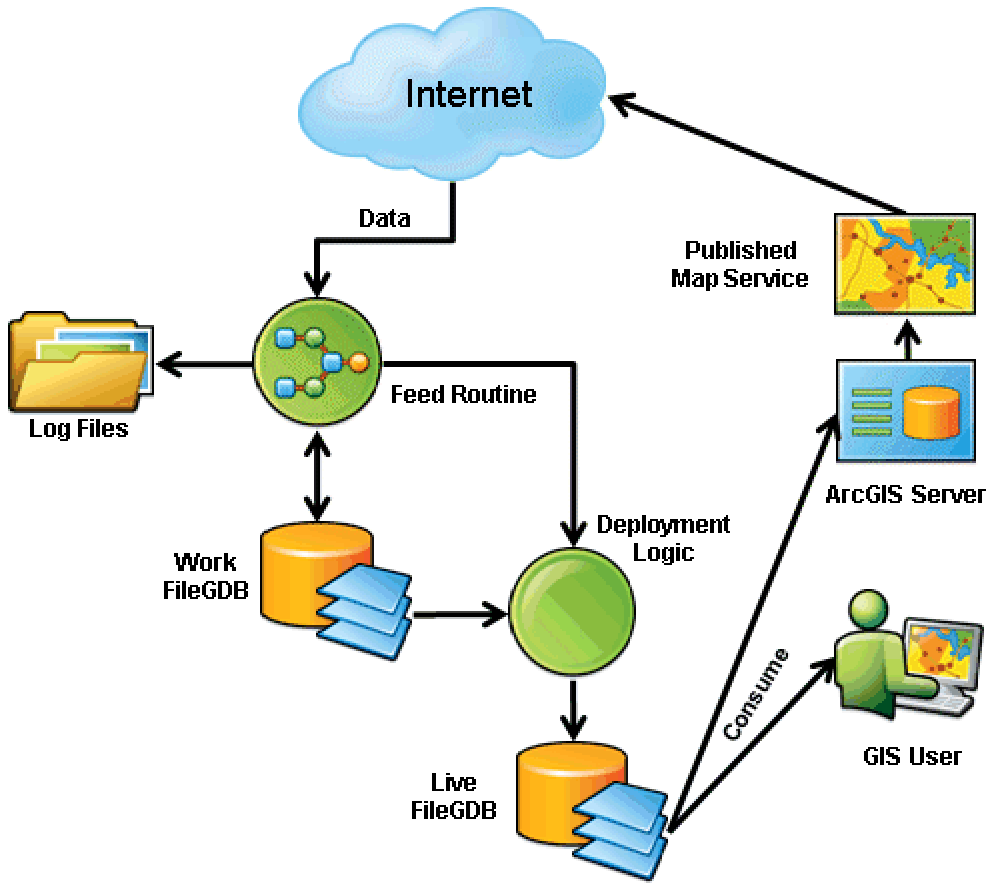
Load ‘B’, while viewing ‘A’



Load ‘A’, while viewing ‘B’

Aggregated Live Feed-Lite methodology

- Using native Python functions, data is downloaded from the source, processed internally, and a private work fGDB Featureclass is replaced or updated using ArcGIS ArcPy functions. See the ‘ALF-Lite’ diagram below.
- Once complete, a deployment function is initiated to provide the administrator the ability to further process the data or deploy it where needed. The work fGDB is copied to a Live folder, where the existing data is replaced using native Python File System copy operations. This allows the data to be replaced very quickly, even though it may be used by a local Desktop user or served by ArcGIS Server.
- Other deployment methods are also possible. For example you could: Package the work fGDB to a Zip file, upload to an Amazon S3 storage bucket, and then download and deploy on an array of ArcGIS Server systems that publish the data through a Map Service.



ALF-Lite Diagram, v1.1

General Guidelines

- Schema and metadata lookup operations are expensive, so ArcGIS clients only perform these actions when the data source is opened or refreshed. Any changes to the schema, metadata, indexes, or overall architecture of the Featureclass made outside of the client connection will NOT be seen until that client reconnects. In fact, some of these changes can even cause client issues or crashes. The only safe changes that can be made while clients are connected are to the Featureclass data.

- The Spatial Extent of a Featureclass is based on the data stored in that Featureclass. As you add or remove features, this extent can change. Since the spatial extent is 'metadata', it will only be read when the client opens the data source. Therefore, the client will not be aware of the extent change when updates are made. This can lead to features that seem to 'disappear' during a pan or zoom operation. Be sure to set the spatial extent of the Featureclass to the maximum extent you will ever expect to see in the data. For the eGDB, simply use the 'sdelayer -o alter -E Xmin,Ymin,Xmax,Ymax' command, which updates the metadata. ~~For the fGDB, you'll need to add 'registration' features, one at the Xmin/Ymin extent and one at the Xmax/Ymax extent.~~ **No longer required or recommended, see below!**

    - ~~An alternative to adding 'registration' features to your Featureclass would be to use a Feature Dataset. Place your Featureclass inside a Feature Dataset and then create a 'Boundary' Featureclass that contains the 'registration' features. The Feature Dataset's Spatial Domain will take on the Extent of its content. Since the 'Boundary' Featureclass has the largest Extent, the Spatial Extent of the Feature Dataset will be set to match. Any Featureclass within the Feature Dataset will also take on this property. Allowing you to control the Spatial Domain without adding ancillary data to your Featureclasses. This should also apply equally well to the eGDB.~~ **No longer required or recommended, see below!**

    - **Please Note:** Issues described above are more often a result of a Spatial Index problem, not likely to have anything to do with the Spatial Extent, see below!

- Spatial Index considerations: The Spatial Index on a Featureclass is extremely important. This index is used whenever a spatial query is performed for an extent of data that is inside or partially inside the Featureclass' Spatial Extent. When a new Featureclass is first created, it does not normally have a Spatial Index defined. This index is typically built immediately following the first set of data rows inserted, or is often rebuild when new rows are added to an existing Featureclass. Trouble is, if this is a Live Featureclass, this addition of rows or replacement of data can cause an auto re-calculation of the Spatial Index size. As you can imagine, this re-calculation will adversely affect the clients that are using this data as they are relying on this index to access the appropriate data rows. This can cause unexpected results similar to the issues described in the Spatial Extent discussion above. To ensure compatibility, **ALWAYS** rebuild the Spatial Index using a predetermined value following an update process and before uploading to the Live data location, never allow ArcGIS to 'pick' or 'calculate' the index grid size. The best tool for rebuilding a Spatial Index is the 'AddSpatialIndex_management' tool found in the ArcPy Data Management Toolbox.

- Avoid running heavy update scripts on systems that also serve data. The update process can further burden a machine by running too many aggregators, causing it to spend more time updating data than serving it! Better to use dedicated systems for aggregating and others for serving.

- When creating the final Featureclasses that will be shared or published, try to use the same coordinate system as the Map Documents you plan to consume them in. This will improve performance by reducing the need to re-project the data on the fly.

- Log History considerations: If your feed routine leverages ArcPy (ArcGIS 10+) or arcgisscripting (pre-ArcGIS 10) Geoprocessing tools to process data, be sure to turn OFF the 'logHistory' option before invoking any Geoprocessing commands. By default, ArcGIS will generate a detailed Log file for each and every Geoprocessing command issued. As a Desktop user, these Logs are maintained and cleaned up automatically. But when run from a script, the Log file maintenance is left up to you to manage. Frequent execution of Geoprocessing commands can generate a significant number of log files over time, taking up a considerable amount of space! Add the following in your script to turn off this option: *(Already included in the ALFprocessor)*

  - For ArcGIS 10+:
    - Where: 'import arcpy'
    - Include: 'arcpy.gp.logHistory = False'

  - For pre-ArcGIS 10:
    - Where: 'import arcgisscripting'
      'gp = arcgisscripting.create()'
    - Include: 'gp.logHistory = False'

- Process Construction considerations: A well-formed Live Feed script should be constructed with the following in mind:

  - It should be as self sufficient as possible.

  - The only outside components it should rely on are those components common to all other Feed scripts (like those that perform a download or handle special processing). *Like the ALFlib and limited functions in the ALFprocessor!*

  - Whenever possible, all output or support content required or maintained by the script should be created by the script itself, not stored as a template or other ancillary component that could be accidentally deleted.

  - The script should be able to run without user intervention.

  - It should be able to run ANYWHERE it is launched, at anytime, regardless of the current state of the process or condition of the environment.

  - It should be as robust as <u>practical</u> and always considering the worst case. You will not be able to trap all conditions, but you should always include a way to report or record the issue (Always use some form of Logging so you can review and trace an issue)! *Supported by ALFlib and ALFprocessor!*

- Time-Enabled content considerations: Time-Enabled Map Layers are often called a Time Series when published to ArcGIS Server. When enabling this content, keep the following in mind:
    - Avoid naming a Date/Time field with the data specific Time Zone, like: ‘UTC_Datetime’. A Time-Enabled layer will include Time Zone details anyway, so there is no need to explicitly name it, and risk confusing users. Also, Clients will often convert the Service Date/Time values to their local time automatically.
    - Always include an Attribute Index on Date/Time fields used in a Time Series. This will help improve performance overall.
    - When a Time-Enabled data set is updated and the Live data is replaced by the ALF Methodology, the updated time intervals will not be reflected in the REST details of the published service. To ensure these details are updated properly, be sure to:
        - Review the layer properties for each Time-Enabled Layer and check the property called ‘Data changes frequently so calculate time extent automatically’. This will enable the periodic data check to gather and report the updated time extent.
        - Flush the Service specific REST cache following a data update. This will force ArcGIS Server to renew the service details on the next service request. Otherwise, the cached details will remain until the service or ArcGIS Server is restarted.
- Performance considerations: Thoughts on improving or managing performance of your feeds.
    - Use ‘in_memory’ resources whenever possible! Writing to or updating disk content can take a considerable amount of time, especially when you need to perform repeated operations. Like creating and calculating fields! Copy a resource to the ‘in_memory’ workspace and then create and calculate your fields, much faster!
    - Keep in mind that if a large number of ‘in_memory’ resources are being created, you will likely run into memory resources issues. Either clean up items periodically or limit usage.
    - Separate your scratch workspace and normal workspace to different physical disk drives if possible to reduce disk I/O contention.
    - Use caution when choosing a Python data type to store lookup items. A List is a great way to store an array of items, but if you need to look up an item by a key name, Lists do not support indexing and therefore will need to compare each

item in the list until it is found. This could take a fair bit of time depending on how large the list is and how often it is being searched. Consider using a Dictionary instead, this supports key name look up operations, much faster!

Leveraging the Enterprise Geodatabase (eGDB)

- For best results, you should always leverage the Esri provided or underlying Database provided Spatial Data Type when creating your Featureclasses. The older style SDE-BINARY storage type stores the spatial data in a separate table, which is joined to the base-table during spatial queries. This can cause client issues when querying the Spatial View after data updates are made, which can lead to application crashes.

- When refreshing (replacing existing data), avoid row-level deletion, this is too expensive and takes far too long to complete. Your users will notice the lack of data and sudden appearance of new data. Better to leverage the Database 'Truncate' operation coupled with a fast append of new data, or simply update each row, for a seamless consumer experience.

- The Feed Routine should connect to the Database with the user that owns or will own the RAW Featureclasses. No other Database user should have access to these RAW Featureclasses. This will prevent other users from blocking the update process and basic administrative functions that need to be performed on the Database tables.

- Use the ArcSDE command-line command 'sdetable -o create_view' to create the Spatial View that all other Database users with eventually access. Create a Database Role that can be granted to other Database users. This will simplify adding and removing user access to the Spatial View. Now you can grant 'select' access on the Spatial View to the Role you created by issuing the command 'sdelayer -o grant'. You can use any other Database tool from this point on to grant or revoke this Role to other users.

Leveraging the File Geodatabase (fGDB)

- About the File Geodatabase or fGDB:
    - The fGDB is a file system folder containing files that represent and store the Geodatabase schema and Featureclass content. Adding or replacing Featureclasses generates new files and updates the internal pointers and metadata used to maintain integrity. Lock files are also created or removed as ArcGIS clients access the file system.
- Though possible, avoid directly updating the fGDB when it is also being accessed by a client. Depending on the operation, some updates can take far too long to complete without causing odd application behaviors like disjointed or missing data. Use a separate workspace and then update the live data using a much faster file system copy operation. If you have no other choice, use row-by-row updates to minimize adverse user experiences. You may also need to set the file system permissions on the Live folder to Read-Only for all but the user running the Feed Routine in order to stop clients from blocking the updates.
- Do not replace an existing Featureclass in your workspace. This will increment the file names in fGDB file structure, leading to file system bloating and data refresh issues. Better to update existing data rows or delete and recreate the fGDB. The underlying file structure needs to remain the same from one update cycle to the next.
- Prior to final deployment, always ‘compact’ the final fGDB before deploying to the Live location. This will clean up the file system and file content, ensuring that the data is as lean and efficient as it can be. This will also reduce bloat by eliminating unnecessary content and shrinking the overall size. It can even improve or stabilize performance.

Leveraging Multiprocessing

In order to keep up with Moore’s Law, many computer manufactures have included Multi-Core and Multi-Socket CPUs in their hardware. This additional CPU power can be leveraged to speed up the processing of heavy-weight data sources or complex workflows that would otherwise be ignored by your typical single Threaded process. Here are some thoughts on how to successfully harness these additional resources to enhance your feeds:

- For the fGDB, multiple processing streams (Threads or Processes) can be employed at the same time as long as they write to their own Featureclass. But, all Featureclasses can reside in the same Feature Dataset or fGDB without conflict! *Multiprocessing is supported by the ALFlib and ALFprocessor!*
    - If more time is spent writing than processing the data, it is best to use a single thread or process for each Featureclass.
    - If more time is spent processing than writing the data, you may be able to dedicate a writer thread or process to handle saving the data while other worker threads or processes handle the bulk processing.
- When an fGDB will have multiple Featureclasses managed by multiple Threads or Processes, NEVER allow the multiprocessing logic to create or overwrite their Featureclasses. Instead, have your initialization process create the fGDB first and then generate each Featureclass to ensure the metadata is always updated correctly and each Geodatabase object is created in the same order each time. Your multi-processing logic should simply append or perform row level data updates.
- Running more than one Feed Routine for different data sets on the same machine can have the same effect as running Multi-Threaded or Multi-Processing for the same Feed. The environment, work spaces and system resources, are the same for all feed processes. To avoid conflict, do not allow your feed routines to share the same work or scratch spaces. Actively set the ‘arcpy.env.scratchWorkspace’ to ‘in_memory’ resources or, use a Feed/Thread/Process specific work folder on disk. Otherwise, you could end up with two different Geoprocessing steps from two different routines that end up picking the same temporary object name, causing them to step on each other’s toes. *Already supported in the ALFprocessor!*

Leveraging Feature Services

A Feature Service or Hosted Feature Service is a great way to leverage the Enterprise ArcGIS Server or ArcGIS On-Line environments. Here are a few thoughts on taking advantage of these:

- Feature Services are managed through web calls to a REST API end-point for each service. Although a Python specific REST API for ArcGIS does not yet exist, you can invoke management calls to secured resources by way of the Python httplib, urllib, or urllib2 libraries. Just takes a little work and patients. *Work is underway to support this directly from the ALFlib!*

- Managing the features within the Feature Service is fairly straight forward when using the ‘addFeature’, ‘deleteFeature’, and ‘updateFeature’ Layer end-points individually. You can also elect to combine various operations by using the ‘applyEdits’ end-point to perform multiple operations in one submission, great when there is only a handful to perform! The only operation that is missing is a truncate, which would be handy for replacing the entire data set when needed. Your only recourse is to perform a ‘deleteFeature’ call with a True condition like ‘1=1’. This is a time consuming and wasteful operation, from the Database perspective, that could delay a Live Feed routine from quickly updating the content. Probably better to use Feature Services for row updates, where there is a change to a known feature. Or where there is a desire to track feature history over time. Could be messy trying to manage old and new data in the same Feature Service, should be considered on a case-by-case basis!

- All-in-all a very convenient way to manage resources.