The GATECloud Paralleliser (GCP)

Large-scale multi-threaded processing with GATE Embedded

version 2.4-SNAPSHOT

Ian Roberts, Valentin Tablan
GATE Team

February 11, 2013

Contents

Chapter 1

Introduction

GCP is a tool designed to support the execution of pipelines built using GATE Developer over large collections of thousands or millions of documents, using a multi-threaded architecture to make the best use of today's multi-core processors. GCP tasks or *batches* are defined using an extensible XML syntax, describing the location and format of the input files, the GATE application to be run, and the kinds of outputs required. A number of standard input and output handlers are provided, but all the various components are pluggable so custom implementations can be used if the task requires it. GCP keeps track of the progress of each batch in a human- and machine-readable XML format, and is designed so that if a running batch is interrupted for any reason it can be re-run with the same settings and GCP will automatically continue from where it left off.

1.1 Definitions

This section defines a number of terms that have specific meanings in GCP.

Batch

A *batch* is the unit of work for a GCP process. It is described by an XML file, and includes the location of a saved GATE application state (a "gapp" file), the location of the *report* file, one *input handler* definition, zero or more *output handler* definitions and a specification of which documents from the input handler should be processed (either as an explicit list of document IDs or as a *document enumerator* which calculates the IDs in an appropriate manner).

Chapter 3 describes the format of batch definition files in detail.

Report

The progress of a running batch is recorded in an XML *report* file as the documents are processed. For each document ID, the report records whether the document was processed successfully or whether processing failed with an error. For successful documents the report includes statistics on how many annotations were found in the document, and for a completed batch it also records overall statistics on the number of documents and total amount of data processed, the total number of successful and failed documents and the total processing time.

The report file for a batch is also the mechanism which allows GCP to recover if processing is unexpectedly interrupted. If GCP is asked to process a batch where the report file already exists it will parse the existing report and ignore documents that are marked as having already been successfully processed. Thus you can simply restart a crashed GCP batch with the same command-line settings and it will continue processing from where it left off on the previous run.

GATE application

A GCP batch specifies the GATE application that is to be run over the documents as a standard "GAPP file" saved application state, which would typically be created using GATE Developer.

Input handler

The *input handler* for a GCP batch specifies the source of documents to be processed. The job of an input handler is to take a document *ID* and load the corresponding GATE `Document` object ready to be processed. There are a number of standard input handlers provided with GCP to take input documents from individual files on disk, directly from a ZIP archive file or from an ARC file as produced by the Heritrix web crawler (`http://crawler.archive.org`). If the standard handlers do not suit your needs then you can provide a custom implementation by including your handler class in a GATE CREOLE plugin referenced by your saved application.

Document enumerator

While the input handler specifies how to go from document IDs to `gate.Document` objects, it does not specify which document IDs are to be processed. The IDs can be specified explicitly in the batch XML file but more commonly an *enumerator* would be used to build a list of IDs by scanning the input directory or archive file. Standard enumerator implementations are provided, corresponding to the standard input handler types, to select a subset of documents from the input directory or archive according to various criteria. As with input handlers, custom enumerator implementations can be provided through the standard CREOLE plugin mechanism.

Output handler

Most batch definitions will include one or more *output handler* definitions, which describe what to do with the document once it has been processed by the GATE application. Standard output handler implementations are provided to save the documents as GATE XML files, plain text and XCES standoff annotations, inline XML ("save preserving format" in GATE Developer terms), and to send the annotated documents to a Mímir server for indexing. Custom implementations can be added using the CREOLE plugin mechanism.

Note that output handler definitions are optional – if you do not specify any output handlers then GCP will not save the results anywhere, but this may be appropriate if, for example, your pipeline contains a custom PR that saves your results to a relational database or similar.

1.2 Processing Model

GCP processes a batch as follows.

1. Parse the batch definition file, and run the document enumerators (if any) to build the complete list of document IDs to be processed.
2. Parse the existing (possibly partial) report file, if one exists, and remove from this list any documents that are already marked as having been successfully processed.
3. Create a thread pool of a size specified on the command line (the default is 6 threads).
4. Load the saved application state, and use `Factory.duplicate` to make additional copies of the application such that there is one independent copy of the application per thread in the thread pool.
5. Run the processing threads. Each thread will repeatedly:
   - take the next available unprocessed document ID from the list.
   - ask the input handler for the corresponding `gate.Document`.
   - put that document into a singleton `Corpus` and run this thread's copy of the GATE application over that corpus.
   - pass the annotated document to each of the output handlers.
   - write an entry to the report file indicating whether the document was processed successfully or whether an exception occurred during processing.
   - release the document using `Factory.deleteResource`.
6. Once all the documents have been processed, shut down the thread pool and call `Factory.deleteResource` to cleanly shut down the GATE applications.

Due to the asynchronous nature of the processing threads, if one document takes a particularly long time to process the other threads can proceed with many other documents in parallel, they are not forced to wait for the slowest thread.

1.3 Changelog

This section summarises the main changes between releases of GCP

1.3.1 2.3 (November 2012)

- Now depends on GATE Embedded 7.1
- Introduced support for *conditional saving of documents* (section 3.3.3)
- Added the *serialized object output handler* (section 3.3.1)
- More robust and reliable counting of the size of each input document.

1.3.2 2.2 (February 2012)

- Now depends on GATE Embedded 7.0
- Introduced Java-based command line interface to replace the `gcp.sh` shell script, which behaves more consistently across platforms.

Chapter 2

Installing and Running GCP

2.1 Installing GCP

There are currently no publicly available binary releases of GCP, so the software must be built from source. The source code is available in the GATE subversion repository. To obtain the current development "trunk", check out

```
http://svn.code.sf.net/p/gate/code/gcp/trunk
```

Specific versions are tagged, and can be checked out from

```
http://svn.code.sf.net/p/gate/code/gcp/tags/<version>
```

To build GCP you will need a Java 6 JDK (update 14 or later). Sun/Oracle, OpenJDK and (on Mac OS X) the Apple JDK have been tested and are known to work. GCJ is known *not* to work. You will also need Apache Ant version 1.7.0 or later. Run "ant distro" to build a ZIP file containing the binary distribution, and unzip that file somewhere to create your GCP installation.

2.2 Running GCP

Once GCP is installed you can run it using the `gcp-cli.jar` executable JAR file in the installation directory (or the `gcp.sh` bash script, which simply calls `java -jar gcp-cli.jar`). This tool takes a number of optional arguments:

-m Specifies the maximum Java heap size, in the format expected by the usual `-Xmx` Java option, e.g. `-m 10G` for a 10GB heap limit. The default setting is `12G`. The `gcp-cli` will spawn a separate java process to run each batch, passing this memory limit to that process. This is different from specifying a `-Xmx` option to `gcp-cli`, which would define the heap size limit for the CLI process itself, not the batch runner processes it spawns.

-t Specifies the number of threads that GCP should use to execute the GATE application. Typically this should be set to between 1 and 1.5 times the number of processing cores available on the machine. The default value is 6, which is generally suitable for a 4-core machine.

-D Java system property settings, for example `-Djava.io.tmpdir=/home/bigtmp`. `-D` options specified before the `-jar` apply to the virtual machine running the CLI, those specified after `-jar gcp-cli.jar` will be passed to the batch runner processes.

The settings for the `-m` and `-t` options are typically a trade-off – if your application is particularly memory-hungry or you are processing particularly large or complex documents you may need to lower the number of processing threads in order to give more memory (on average) to each one.

GCP can run in two modes. In the basic "single-batch" mode the final command-line argument is simply the path to a single *batch definition* XML file (see chapter 3 for details), and GCP will process that batch and then exit.

The other (and more commonly used) mode is "multi-batch" mode, signified by the `-d` command line option. In this mode the final command-line argument is the path to a directory referred to as the *working directory.*

```
java -jar gcp-cli.jar -t 4 -m 8G -d /data/gcp
```

The working directory is expected to contain a subdirectory named "in", and any file in this directory with the extension `.xml` (in lower case) is assumed to be a batch definition file. For each batch *batch*`.xml` in the "in" directory, the script will:

- run the batch, redirecting the standard output and error streams to a file *working-dir*`/logs/`*batch*`.xml.log`
- if the batch completes successfully, move the definition file to *working-dir*`/out/`*batch*`.xml`
- or, if the batch fails (i.e. the Java process exits with a non-zero exit code, which occurs if, for example, one of the processing threads encounters an OutOfMemoryError), move the definition file to *working-dir*`/err/`*batch*`.xml`

Additional batches can be added to the "in" directory at any time – whenever a batch completes the script will re-scan the "in" directory to locate the next available batch. In particular, failed batches can be moved back from "err" to "in" and they will be re-processed, and if the report file for the failed batch is intact GCP will continue on from where it left off on the previous run.

Creating a file named `shutdown.gcp` in the "in" directory will cause the script to exit at the end of the batch it is currently processing (or immediately if it is currently idle).

Chapter 3

The Batch Definition File

3.1 The Structure of a Batch Descriptor

GCP batches are defined by an XML file whose format is as follows. The root element defines the batch identifier:

```
<batch id="batch-id" xmlns="http://gate.ac.uk/ns/cloud/batch/1.0">
```

The children of this `<batch>` element are:

application (required) specifies the location of the saved GATE application state.
`<application file="../annie.xgapp"/>`

report (required) specifies the location of the XML report file. If the report file already exists GCP will read it and process only those documents that have not already been processed successfully. `<report file="../report.xml" />`

input (required) specifies the input handler which will be the source of documents to process.

output (zero or more) specified what to do with the documents once they have been processed.

documents (required) specifies the document IDs to be processed, as any combination of the child elements:

- **id** a single document ID. `<id>bbc/article001.html</id>`
- **documentEnumerator** an enumerator that generates a list of IDs. The enumerator implementation chosen will typically depend on the specific type of input handler that the batch uses.

The following example shows a simple XML batch definition file which runs ANNIE and saves the results as GATE XML format. The input, output and documents elements are discussed in more detail in the following sections.

```
<?xml version="1.0" encoding="UTF-8"?>
<batch id="sample" xmlns="http://gate.ac.uk/ns/cloud/batch/1.0">
  <application file="../annie.xgapp"/>

  <report file="../reports/sample-report.xml" />

  <input dir="../input-files"
         mimeType="text/html"
         compression="none"
         encoding="UTF-8"
         class="gate.cloud.io.file.FileInputHandler" />

  <output dir="../output-files-gate"
          compression="gzip"
          encoding="UTF-8"
          fileExtension=".GATE.xml.gz"
          class="gate.cloud.io.file.GATEStandOffFileOutputHandler" />

  <documents>
    <id>ft/03082001.html</id>
    <id>gu/04082001.html</id>
    <id>in/09082001.html</id>
  </documents>
</batch>
```

It is important to note that all relative file paths specified in a batch descriptor are resolved against the location of the descriptor file itself, thus if this descriptor file were located at `/data/gcp/batches/sample.xml` then it would load the application from `/data/gcp/annie.xgapp`.

3.2 Specifying the Input Handler

Each batch definition must include a single `<input>` element defining the source of documents to be processed. Given a document ID, the job of the input handler is to locate the identified document and load it as a `gate.Document` to be processed by the application. Note that the input handler describes how to find the document for each ID but does *not* define which IDs are to be processed, that is the job of the `<documents>` element below.

The `<input>` element must have a `class` attribute specifying the name of the Java class implementing the handler. GCP will create an instance of this class and pass the remaining `<input>` attributes to the handler to allow it to configure itself. Thus, which attributes are supported and/or required depends on the specific handler class.

GCP provides three standard input handler types:

- `gate.cloud.io.file.FileInputHandler` to read documents from individual files on the filesystem
- `gate.cloud.io.zip.ZipInputHandler` to read documents directly from a ZIP archive
- `gate.cloud.io.arc.ARCInputHandler` to read documents from an ARC archive as produced by the Heritrix web crawler (`http://crawler.archive.org`).

3.2.1 The `FileInputHandler`

`FileInputHandler` reads documents from individual files on the filesystem. It can read any document format supported by GATE Embedded, and in addition it can read files that are GZIP compressed, unpacking them on the fly as they are loaded. It supports the following attributes on the `<input>` element in the batch descriptor:

encoding (optional) The character encoding that should be used to read the documents (i.e. the value for the encoding parameter when creating a `DocumentImpl` using the GATE Factory). If omitted, the default GATE Embedded behaviour applies, i.e. the platform default encoding is used.

mimeType (optional) The MIME type that should be assumed when creating the document (i.e. the value of the `DocumentImpl` mimeType parameter). If omitted GATE Embedded will attempt to guess the appropriate MIME type for each document in the usual way, based on the file name extension and magic number tests.

compression (optional) The compression that has been applied to the files, either "none" (the default) or "gzip".

The actual mapping from document IDs to file locations is controlled by a *naming strategy*, another Java object which is configured from the `<input>` attributes. The default naming strategy (`gate.cloud.io.file.SimpleNamingStrategy`) treats the document ID as a relative path[1], and takes the following attributes:

dir (required) The base directory under which documents are found.

fileExtension (optional) A file extension to append to the document ID.

Given a document ID such as "ft/03082001", a base directory of "/data" and a file extension of ".html" the `SimpleNamingStrategy` would load the file "/data/ft/03082001.html"

To use a different naming strategy implementation, specify the Java class name of the custom strategy class as the `namingStrategy` attribute of the `<input>` element, along with any other attributes the strategy requires to configure it.

3.2.2 The `ZipInputHandler`

The ZIP input handler reads documents directly out of a ZIP archive, and is configured in a similar way to the file-based handler. It supports the following attributes:

encoding (optional) exactly as for `FileInputHandler`

mimeType (optional) exactly as for `FileInputHandler`

zipFile (required) The location of the ZIP file from which documents will be read.

fileNameEncoding (optional) The default character encoding to assume for file names inside the ZIP file. This attribute is only relevant if the ZIP file contains files whose names contain non-ASCII characters *without* the "language encoding flag" or "Unicode extra fields", and can be omitted if this does not apply. There is a detailed discussion on file name encodings in ZIP files in the Ant

[1]Technically a relative *URI*, so forward slashes must be used in document IDs even when running on Windows where file paths normally use backslashes.

manual (`http://ant.apache.org/manual/Tasks/zip.html#encoding`), but the rule of thumb is that if the ZIP file was created using Windows "compressed folders" then `fileNameEncoding` should be set to match the encoding of the machine that created the ZIP file, otherwise the correct value is probably "Cp437" or "UTF-8".

The ZIP input handler does not use pluggable naming strategies, and simply assumes that the document ID is the path of an entry in the ZIP file.

3.2.3 The `ARCInputHandler`

The ARC input handler reads documents out of an Internet Archive ARC file. It supports the following attributes:

arcFile (required) The location of the ARC file.

defaultEncoding (optional) The *default* character encoding to assume for ARC entries that do not specify their encoding in the entry headers. If an entry specifies its own encoding explicitly this will be used. If this attribute is omitted, "Windows-1252" is assumed as the default.

mimeType (optional) The MIME type that should be assumed when creating the document (i.e. the value of the `DocumentImpl` mimeType parameter). If omitted, the MIME type specified by the ARC entry will be used, if present, and if the entry does not specify a MIME type header then the usual GATE Embedded heuristics will apply.

The ARC input handler expects its document IDs to begin with one or more digits (everything from the first non-digit character in the ID is ignored). These leading digits are treated as a zero-based index into the ARC file, i.e. any of the IDs "1", "000001" or "000001_http://example.com" are treated as referring to the second entry in the archive (0 would be the first entry).

The ARC input handler adds all the HTTP headers and ARC record headers for the entry as features on the GATE `Document` it creates. HTTP header names are prefixed with "http_header_" and ARC record headers with "arc_header_".

3.3 Specifying the Output Handlers

Output handlers are responsible for taking the GATE Documents that have been processed by the application and doing something with the results. GCP supplies a number of standard output handlers to save the document text and annotations to files in various formats, and also a handler to send the annotated documents to a remote Mímir server for indexing.

Most batches would specify at least one output handler but GCP does support batches with no outputs (if, for example, the application itself contains a PR responsible for outputting results).

Output handlers are specified using `<output>` elements in the batch definition, and like input handlers these require a `class` attribute specifying the implementing Java class name. Other attributes are passed to the instantiated handler object to allow it to configure itself.

By default, an output handler will save all annotations from all annotation sets in each document. A given output handler may be configured to save only a subset of the annotations by providing `<annotationSet>` sub-elements inside the `<output>` element, for example

```
1  <annotationSet name="ANNIE">
2    <annotationType name="Person" />
3    <annotationType name="Location" />
4  </annotationSet>
5
6  <annotationSet />
```

The `<annotationSet>` element may have a `name` attribute giving the annotation set name (if omitted the default annotation set is used), and zero or more `<annotationType>` sub-elements giving the annotation types to extract from that set (if no `<annotationType>` elements are provided, all annotation from the set are saved, so line 6 specifies that the handler should save all annotations from the default set).

Note that these filters are provided as a convenience, and some output handler implementations may ignore them. For example the Mímir output handler always sends the complete Document to the Mímir server, regardless of the filters specified.

3.3.1 File-based Output Handlers

GCP provides a set of five standard file-based output handlers to save data to files on the filesystem in various formats.

- `gate.cloud.io.file.GATEStandOffFileOutputHandler` to save documents in the GATE XML format ("save as XML" in GATE Developer).
- `gate.cloud.io.file.GATEInlineOutputHandler` to save documents with inline XML tags for their annotations ("save preserving format" in GATE Developer).
- `gate.cloud.io.file.PlainTextOutputHandler` to save just the text content of the document. This is rarely useful on its own but is frequently used in conjunction with
- `gate.cloud.io.xces.XCESOutputHandler` to save annotations in the XCES standoff format. Annotation offsets in XCES refer to the plain text as saved by a `PlainTextOutputHandler`.
- `gate.cloid.io.file.SerializedObjectOutputHandler` to save documents using Java's built in *object serialization* protocol (with optional compression). This handler ignores annotation filters, and always writes the complete document. This is the same mechanism used by GATE's `SerialDataStore`.

The five handlers share the following `<output>` attributes:

encoding (optional, not applicable to `SerializedObjectOutputHandler`) The character encoding used when writing files. If omitted, "UTF-8" is the default.

compression (optional) The compression algorithm to apply to the saved files. Can be either "none" (no compression, the default) or "gzip" (GZIP compression).

As with the file-based input handler, these output handlers use a *naming strategy* to map from document IDs to output file names. The default strategy is the same `SimpleNamingStrategy` configured with a base `dir` and a `fileExtension`, treating the document ID as a path relative to the given directory and appending the given extension. This is appropriate when using a file or ZIP input handler but for batches that use an `ARCInputHandler` a different strategy is required.

As document IDs for an `ARCInputHandler` are simple numbers (with an optional suffix) the simple strategy would put all the output files into a single directory. Directories with very large numbers of files can lead to poor performance on many filesystems, so an alternative strategy is provided that left-pads the document ID numbers with zeros and puts them into a hierarchy of directories. To use this strategy, specify an attribute `namingStrategy="gate.cloud.io.arc.ARCDocumentNamingStrategy"`, and the usual `dir` and `fileExtension` attributes of the default strategy. The ARC strategy also accepts an optional additional attribute `pattern` defining the pattern to use to map the ID number to a directory.

The default pattern is "3/3", which will left-pad the ID to a minimum of 6 digits and then create one level of directories from the first three digits and use the last three as part of the file name[2]. The trailing characters of the document ID after the numeric index are cleaned up to replace slash and colon characters with underscores (so the resulting file name will not include any more levels of subdirectories). For full details of this process, see the JavaDoc documentation. As an example, the ID "001_http://example.com/file.html" with the default pattern of "3/3" would map to the target path "000/001_example.com_file.html", and this would then be combined with the `dir` and `fileExtension` to produce the final file name.

The `PlainTextOutputHandler` simply saves the plain text of the GATE document with no annotations (so `<annotationSet>` filters are ignored). The `GATEStandOffFileOutputHandler` writes the document text and selected annotations in the standard "save as XML" GATE XML format. The `XCESOutputHandler` saves the selected annotations as XCES XML format.

The `GATEInlineOutputHandler` saves the document text plus selected annotations as inline XML tags as produced by "save preserving format" in GATE Developer. This handler supports one additional `<output>` attribute named `includeFeatures` – if this is set to "true", "yes" or "on" then the annotation features will be included as attributes on the XML tags, otherwise (including if the attribute is omitted) it will save just the tags with no attributes.

[2]In fact the pattern is processed from right to left, so any surplus digits end up in the first place, i.e. the ID 1234567 becomes 1234/567 rather than 123/4567.

3.3.2 The Mímir Output Handler

GCP also provides `gate.cloud.io.mimir.MimirOutputHandler` to send annotated documents to a Mímir server for indexing. This handler supports the following `<output>` attributes:

indexUrl (required) the *index URL* of the target index. See the Mímir documentation for details.

uriFeature (optional) Mímir requires a URI to identify each document. This attribute tells GCP that the URI for a document should be taken from the document feature with this name.

namespace (optional) if `uriFeature` is not specified GCP will construct a suitable URI by appending the document ID to a fixed "namespace" string. If omitted an empty namespace will be used (i.e. the URI passed to Mímir will be just the document ID).

username (optional) HTTP basic authentication username to pass to the Mímir index. If omitted, no authentication token will be passed.

password (required if and only if username is specified) the corresponding basic authentication password.

This handler ignores annotation set filters – the complete document will be sent to Mímir.

3.3.3 Conditional Output

All output handlers support conditional output: the option to only save some of the documents, based on the value of a document feature. To make use of this facility, you need to specify which document feature should be read when deciding whether or not to save a given document, by adding an attribute named `conditionalSaveFeatureName` to the `output` XML tag in the batch definition, like in the following example:

```
<output conditionalSaveFeatureName="save"
    dir="../output-files-gate"
    compression="gzip"
    encoding="UTF-8"
    fileExtension=".GATE.xml.gz"
    class="gate.cloud.io.file.GATEStandOffFileOutputHandler" />
```

In this example, for each processed document, a document feature named `save` will be sought. If this is found, and if its value is logical true (i.e. the feature value is a String with the content '*true*', '*yes*', or '*on*', regardless of case) then the document will be saved, otherwise it will be ignored.

3.4 Specifying the Documents to Process

The final section of the batch definition specifies which document IDs GCP should process. The IDs can be specified in two ways:

- Directly in the XML as `<id>doc/id/here</id>` elements.
- By defining a *document enumerator* which generates a list of IDs from some external source.

GCP provides document enumerator implementations corresponding to the default input handlers, so a typical batch with a ZIP input handler, for example, would use a ZIP enumerator (configured for the same ZIP file) to generate the list of document IDs.

A document enumerator is configured using a `<documentEnumerator>` XML element inside the `<documents>` element. As with input and output handlers, this element requires a `class` attribute specifying the Java class of the enumerator implementation and other attributes are handed off to the enumerator object to configure itself.

3.4.1 The File and ZIP enumerators

The default enumerator implementation corresponding to the file and ZIP input handlers are closely related to one another.

The `gate.cloud.io.file.FileDocumentEnumerator` takes a `dir` attribute and the `gate.cloud.io.zip.ZipDocumentEnumerator` takes `zipFile` and `fileNameEncoding` attributes (as described above for their corresponding input handlers) specifying where to find the directory or ZIP file to be enumerated. To define which files (or ZIP entries) to enumerate, the enumerators use the "fileset" abstraction from Apache Ant, controlled by the following attributes:

includes (optional) comma-separated file name patterns specifying which files to include in the search, e.g. `"**/*.xml,**/*.XML"`. If omitted, all files or ZIP entries are included.

excludes (optional) comma-separated file name patterns specifying which files to exclude from the search, e.g. `"**/*.ignore.xml"`. If omitted, nothing is excluded.

defaultExcludes (optional) Ant filesets exclude certain file patterns by default (`http://ant.apache.org/manual/dirtasks.html#defaultexcludes`), and the GCP enumerators behave likewise. To *disable* the default excludes, set this attribute to "off" or "no".

prefix (optional) a prefix to prepend to the paths that are returned by the fileset. For example, if a batch has a file input handler pointing to the directory `/data` and an enumerator pointing to the directory `/data/large` then the enumerator would need a prefix of "large/" to produce IDs that are meaningful to the input handler.

See the Ant documentation for full details on the include and exclude patterns supported by filesets. The IDs returned by the enumerator will be those that match at least one of the include patterns and also do not match any of the exclude patterns.

Note also that include and exclude patterns are *case sensitive*, so a pattern of "*.xml" would not match "FILE.XML", for example. To match both upper and lower-case variants, include both forms in the pattern.

3.4.2 The ARC enumerator

The `gate.cloud.io.arc.ARCDocumentEnumerator` enumerates entries in an ARC file, and would typically be used in conjunction with an ARC input handler. The enumerator supports the following attributes:

arcFile (required) the path to the ARC archive to enumerate.

mimeTypes (optional) whitespace-separated list of MIME types. If specified, the enumerator will only include entries in the ARC file whose header specifies one of the given MIME types. So a value of `"text/html application/pdf"` would enumerate only HTML and PDF files from the archive.

includeStatusCodes (optional) Each entry in an ARC file records the HTTP status code (200, 301, 404, etc.) that was returned by the server when the item was crawled. This attribute gives a regular expression that is matched against the status codes that should be included in the enumeration. If omitted, all status codes are included (except those excluded by `excludeStatusCodes`).

excludeStatusCodes (optional) regular expression giving the status codes that should be excluded from the enumeration. If *both* `includeStatusCodes` *and* `excludeStatusCodes` are omitted, the default behaviour is to assume an exclude pattern of `[345].*` (i.e. omit all 3xx, 4xx and 5xx status codes).

The ARC enumerator returns document IDs of the form "nnnnnn_*entryURL*", i.e. the zero-based index into the archive, left-padded with zeros to a minimum of 6 digits, followed by an underscore, followed by the original URL of the archive entry, e.g. `000004_http://example.com/file.html`. This format is designed to work well in combination with the `ARCDocumentNamingStrategy` for file-based output handlers.

3.4.3 The `ListDocumentEnumerator`

`gate.cloud.io.ListDocumentEnumerator` is the final enumerator implementation provided by default, and it simply reads a list of document IDs from a plain text file, one ID per line. It supports the following attributes:

file (required) the location of the text file.

encoding (optional) the character encoding of the file. If omitted, the platform default encoding is assumed.

prefix (optional) a common prefix to prepend to each ID (see the file-based enumerator for an example of this).

This enumerator treats each line of the specified file as a separate document ID, ignoring leading and trailing whitespace on the line. It is intended for use when there is a separate process (such as a Perl script) that generates the list of IDs in advance.

Chapter 4

Extending GCP

GCP has been designed to be easily extensible. Additional input/output handlers, naming strategies and document enumerators can be added by writing a Java class that implements the relevant interface and placing that class in a CREOLE plugin loaded by your application. The class can then be named in your batch definitions as appropriate, and GCP will load it from the GATE ClassLoader.

4.1 Custom Input Handlers

Input handlers must implement the `gate.cloud.io.InputHandler` interface:

```
public void config(Map<String, String> configData) throws
    IOException, GateException;
public void init() throws IOException, GateException;
public void close() throws IOException, GateException;

public Document getInputDocument(String id) throws IOException,
    GateException;
```

The handler class must also have a no-argument constructor. When GCP parses the batch definition it creates an instance of the handler class using that constructor, then calls the `config` method, passing in a map containing the XML attribute values from the `<input>` element in the batch file. An additional "virtual" attribute named `batchFileLocation`[1] is also available in this map, containing the path to the XML batch definition file itself. This is made available to allow the handler to resolve relative path expressions in other attribute values against the location of the XML file (see the standard handler implementations for examples of this). The `config` method should read any required parameters from the config map, and should throw a suitably descriptive `GateException` if any required parameters are missing.

Next, GCP will call the handler's `init()` method. This method is responsible for setting up the handler, and is where you should open any required external resources

[1]Available as a constant `gate.cloud.io.IOConstants.PARAM_BATCH_FILE_LOCATION`

such as index files. Both `config` and `init` are guaranteed to be called sequentially in a single thread.

Conversely, `getInputDocument` will be called by the processing threads, and must therefore be thread-safe. However you should avoid locking or `synchronized` blocks within `getInputDocument` if at all possible, so as not to block processing threads against one another. This method is the one responsible for actually loading the GATE Document corresponding to a given document ID. The handler does not need to (indeed should not) retain a reference to the document that it returns, as the processing thread is responsible for freeing the document with `Factory.deleteResource` when processing is complete.

Finally, the `close` method will be called (in a single thread) when the entire batch is complete. This is where you should release resources that were acquired in the `init` method.

It is good manners, though not strictly required, for input handlers to provide a meaningful `toString()` implementation.

4.2 Custom Output Handlers

Output handlers must implement the `gate.cloud.io.OutputHandler` interface:

```
public void setAnnSetDefinitions(List<AnnotationSetDefinition>
    annotationsToSave);
public List<AnnotationSetDefinition> getAnnSetDefinitions();

public void config(Map<String, String> configData) throws
    IOException, GateException;
public void init() throws IOException, GateException;
public void close() throws IOException, GateException;

public void outputDocument(Document document, String documentId)
    throws IOException, GateException;
```

GCP will instantiate the handler class and call `config` exactly as for input handlers (see section 4.1). It will then call `setAnnSetDefinitions` to pass in the annotation set definitions specified by the `<annotationSet>` elements in the XML (if any), and then call `init`. As before these three methods are called in sequence in a single thread.

As documents are processed, the various processing threads will call `outputDocument`, passing an annotated document as a parameter. This method must therefore be thread-safe, but should avoid synchronization and locking if at all possible.

Finally, the `close` method will be called (in a single thread) when the entire batch is complete. This is where you should release resources that were acquired in the `init` method.

It is good manners, though not strictly required, for output handlers to provide a meaningful `toString()` implementation.

GCP provides an abstract base class `gate.cloud.io.AbstractOutputHandler` which custom output handler implementations may choose to extend. This class provides an implementation of the `get` and `setAnnSetDefinitions` methods, and a utility method to extract the annotations corresponding to these definitions from an annotated document at output time.

4.3 Custom Naming Strategies

Custom naming strategies (for use with the file-based input and output handlers) must implement the `gate.cloud.io.file.NamingStrategy` interface:

```java
public void config(boolean isOutput, Map<String, String>
    configData) throws IOException, GateException;

public File toFile(String id) throws IOException;
```

The `config` method is similar to the equivalent method on input and output handlers, but it has an extra parameter indicating whether this naming strategy is being used by an input (false) or output (true) handler. This allows the strategy to adjust its behaviour as appropriate, for example in the input case it is an error if the specified base directory does not exist, whereas for output this is permitted as the output handler will create intermediate directories as required.

The `toFile` method will be called from processing threads by the input (or output) handler that uses this strategy, and should return the `java.io.File` that corresponds to the given document ID.

4.4 Custom Document Enumerators

Document enumerators must implement the `gate.cloud.io.DocumentEnumerator` interface:

```java
public void config(Map<String, String> configData) throws
    IOException, GateException;
public void init() throws IOException, GateException;

// DocumentEnumerator extends java.util.Iterator<String>
public boolean hasNext();
public String next();
public void remove();
```

They have the familiar `config` and `init` methods which are called as for input and output handlers (see section 4.1). To actually enumerate documents GCP uses the standard `Iterator<String>` methods, the enumerator is expected to return one ID from each call to `next`. The `remove` method is specified by the Iterator interface but will never be called by GCP, the standard enumerators all implement this method to throw an `UnsupportedOperationException`.

Chapter 5

Advanced Topics

5.1 GATE Configuration

GCP uses its own GATE site and user configuration files, located in the `gate-home` directory under the GCP installation root directory. It deliberately does not use the user's own `~/.gate.xml`, so if you need to configure any options that must be set through configuration files you will need to edit `gate-home/user-gate.xml`. The most likely configuration option that you will need to set this way is whether or not to add additional spaces to separate otherwise adjacent text in different XML or HTML elements when unpacking markup. This is the `Document_add_space_on_unpack` option in `user-gate.xml`.

5.2 JMX Monitoring

The GCP batch runner registers an MBean with the platform JMX MBean server in its JVM that makes it possible to query the state of the running batch from the JMX management console or (using the standard JMX APIs) from another Java process. The process of connecting a JMX client to the GCP process is beyond the scope of thie guide, here we simply describe the MBean interface and the attributes it exposes to clients.

The MBean implements the `gate.cloud.batch.BatchJobData` interface:

```java
public JobState getState();
public int getProcessedDocumentCount();
public int getTotalDocumentCount();
public int getRemainingDocumentCount();
public int getSuccessDocumentCount();
public int getErrorDocumentCount();
public long getStartTime();
public String getBatchId();
public long getTotalDocumentLength();
public long getTotalFileSize();
```

The `getState()` method returns the current state of the batch. The state is an enum type, and will usually be `JobState.RUNNING`. The various `get*DocumentCount` methods provide access to the number of documents that have been processed, the number of those that were successful/failed and the number remaining to be processed. This allows the monitoring process to check that the GCP process is not "stuck". The `getTotalDocumentLength` and `getTotalFileSize` methods give the total length of the documents processed (in plain text characters, after GATE has unpacked the markup) and the total size (in bytes) of the files processed so far.