UIMA Asynchronous Scaleout

Written and maintained by the Apache UIMA Development Community

Version 2.3.1

Publication date March, 2011

Table of Contents

1. Overview - Asynchronous Scaleout 1
    1.1. Terminology 1
    1.2. AS versus CPM 2
    1.3. Design goals for Asynchronous Scaleout 3
    1.4. AS Concepts 4
        1.4.1. Threading 4
        1.4.2. AS Component wrapping 4
        1.4.3. Parallel Flows 6
        1.4.4. Deployment alternatives 6
        1.4.5. Limits 8
        1.4.6. Compatibility 8
    1.5. Application Concepts 8
        1.5.1. Application API 9
        1.5.2. Collection Process Complete 9
    1.6. Monitoring & Controlling 9
    1.7. JMS Service Descriptor 9
    1.8. Life cycle 10
2. Error Handling for Asynchronous Scaleout 11
    2.1. Basic concepts 11
    2.2. Associating Errors with incoming commands 11
        2.2.1. Error handling - CAS Multipliers 12
    2.3. Error handling overview 12
    2.4. Error results 14
    2.5. Error Recovery actions 14
        2.5.1. Aggregate Error Actions 14
    2.6. Thresholds for Terminate and Disable 15
    2.7. Terminate Action 15
    2.8. Commands and allowed actions 16
3. Asynchronous Scaleout Deployment Descriptor 17
    3.1. Descriptor Organization 17
    3.2. Deployment Descriptor 17
    3.3. CAS Pool 18
    3.4. Service 18
    3.5. Customizing the deployment 19
    3.6. Input Queue 19
    3.7. Top Level AE Descriptor 20
    3.8. Setting Environment Variables 20
    3.9. Analysis Engine 20
    3.10. Error Configuration descriptors 23
    3.11. Error Configuration defaults 24
4. Asynchronous Scaleout Application Interface 27
    4.1. Async Client API 27
    4.2. The UimaAsynchronousEngine Interface 27
    4.3. Application Context Map 29
    4.4. Status Callback Listener 29
    4.5. Error Results 30
    4.6. Asynchronous Client API Usage Scenarios 30
        4.6.1. Instantiating a Client API Object 30
        4.6.2. Calling an Existing Service 30
        4.6.3. Retrieving Asynchronous Results 31
        4.6.4. Deploying a Service with the Client API 31

4.7. Undeploying a Service with the Client API 32
4.8. Recovering from broker failure 32
4.9. Sample Code 32
5. Monitoring, Tuning and Debugging 33
5.1. Monitoring 33
5.1.1. JMX 33
5.1.2. JMX Information from UIMA AS 34
5.2. Logging Sampled JMX information at intervals 39
5.2.1. Configuring JVM to run the monitor 39
5.2.2. Running the Monitor program standalone 40
5.2.3. Monitoring output 40
5.3. Tuning 41
5.3.1. Tuning procedure 41
5.3.2. Tuning Settings 42
5.4. Debugging 42
5.4.1. Error Reporting and Tracing 43
5.4.2. CAS Logging 43
6. Asynchronous Scaleout Camel Driver 45
6.1. Overview 45
6.2. How does it work? 45
6.3. URI Format 45
6.4. Sample 46
6.5. Implementation 47

Chapter 1. Overview - Asynchronous Scaleout

UIMA Asynchronous Scaleout (AS) is a set of capabilities supported in the UIMA Framework for achieving scaleout that is more general than the approaches provided for in the Collection Processing Manager (CPM). AS is a second generation design, replacing the CPM and Vinci Services. The CPM and Vinci are still available and are not being deprecated, but new designs are encouraged to use AS for scalability, and current designs reaching limitations may want to move to AS.

AS is integrated with the flow controller architecture, and can be applied to both primitive and aggregate analysis engines.

1.1. Terminology

Terms used in describing AS capabilities include:

AS
Asynchronous Scaleout - a name given to the capability described here

AS-JMS/AMQ/Spring
A variety of AS, based on JMS (Java Messaging Services), Active MQ, an Apache Open Source implementation of JMS, and the Spring framework. This variety is the one described in detail in this document.

Queue
Queues are the basic mechanism of asynchronous communication. One or more "producers" send messages to a queue, and a queue can have one or more "consumers" that receive messages. Messages in UIMA AS are usually CASes, or references to CASes. Some queues are simple internal structures; others are JMS queues which are identified by a 2 part name: the first part is the Queue Broker; the second part is a Queue Name.

AS Component
An AS client or service. AS clients send requests to AS service queues and receive back responses on reply queues. AS services can be AS Primitives or AS aggregates (see following).

AS Primitive
An AS service that is either a Primitive Analysis Engine or an Aggregate AE whose Delegates are **not** AS-enabled

AS Aggregate
An AS service that is an Aggregate Analysis Engine where the Delegates are also AS components.

AS Client
A component sending requests to AS services. An AS client is typically an application using the UIMA AS client API, a JMS Service Client Proxy, or an AS Aggregate.

co-located
two running pieces of code are co-located if they run in the same JVM and share the same UIMA framework implementation and components.

Queue Broker
Queue brokers manage one or more named queues. The brokers are identified using a URL, representing where they are on the network. When the queue broker is co-located with the AS client and service, CASes are passed by reference, avoiding serialization / deserialization.

Transport Connector
AS components connect to queue brokers via transport connectors. UIMA AS will typically use "tcp" connectors. "http" connectors are also available, and are useful for tunneling through firewalls via an existing public web server.

1.2. AS versus CPM

It is useful to compare and contrast the approaches and capabilities of AS and CPM.

	AS	CPM
Putting components together	Provides a consistent, single, unified way to put components together, using the base UIMA "aggregate" capability.	**Two methods of putting components together** 1. CPE (Collection Processing Engine) descriptor, which has sections specifying a Collection Reader, and a set of CAS Processors 2. Each CAS Processor can, as well, be an aggregate
Kinds of Aggregates	An aggregate can be run **asynchronously** using the AS mechanism, with a queue in front of each delegate, or it can by run **synchronously**. When run asynchronously, *all* of the delegates will have queues in front of them, and delegates which are AS Primitives can be individually scaled out (replicated) as needed. Also, multiple CASes can be in-process, at different steps in the pipeline, even without replicating any components.	All aggregates are run synchronously. In an aggregate, only one component is running at a time; there is only one CAS at a time being processed within the aggregate.
CAS flow	Any, including custom user-defined sequence using user-provided flow controller. Parallel flows are supported.	Fixed linear flow between CAS processors. A single CAS processor can be an aggregate, and within the aggregate, can have any flow including custom user-defined sequence using user-provided flow controller.
Threading	Each instance of a component runs in its own thread; the same thread used to call `initialize()` for a particular instance of a component is used when calling `process()`.	One thread for the collection reader, one for the CAS Consumers, "n" threads for the main pipeline, with no guarantees that the same thread for the `initialize()` call is used for the `process()` call.
Delegate deployment	Co-located or remote.	Co-located or remote.
Life cycle management	Scripts to launch services, launch Queue Brokers.	Scripts to launch services, start Vinci Name Service. In addition, CPE "managed" configuration provides for automatic launching of UIMA

	AS	CPM
		Vinci services in same machine, in different processes.
Error recovery	Similar capabilities as the CPM provides for CAS Processors, but at the finer granularity of each AS component. The support includes customizable behavior overrides and extensions via user code.	Error detection, thresholding, and recovery options at the granularity of CAS Processors (which are CPM components, not delegates of aggregates), with some customizable callback notifications
Firewall interactions	Enables deployment of AS services behind a firewall using a public broker. Enables deployment of a public broker through single port, or using HTTP "tunneling".	When using Vinci protocol, requires opening a large number of ports for each deployed service. SOAP connected services require one open port.
Monitoring and Tuning	JMX (Java Management Extensions) are enabled for recording many kinds of statistical information, and can be used to monitor (and control) the operations of AS configured systems. Statistics are provided and summarized from remote delegates, to aid in tuning scaled-out deployments.	Some JMX information
Collection Reader	Supported for backwards compatibility. New programs should use the CAS Multiplier instead, which is more general, or have the application pass in CASes to be processed. The compatibility support wraps Collection Readers as Cas Multipliers. Note: this is supported and implemented in base UIMA.	Is always first element in linear CPE sequence chain

1.3. Design goals for Asynchronous Scaleout

The design goals for AS are:

1. Increased flexibility and options for scaleout (versus CPM)
    a. scale out parts independently of other parts, to appropriate degree
    b. more options for protocols for remote connections, including some that don't require many ports through firewalls
    c. support multiple CASes in process simultaneously within an aggregate pipeline
2. Build upon widely accepted Apache-licensed open source middleware
3. Simplification:
    a. Standardize on single approach to aggregate components
    b. More uniform Error handling / recovery / monitoring for all AS managed components.
    c. No changes to existing annotator code or descriptors. An additional deployment descriptor is used to augment the conventional descriptors.

1.4. AS Concepts

1.4.1. User written components and multi-threading

AS provides for scaling out of annotators - both aggregates and primitives. Each of these can specify a user-written implementation class. For primitives, this is the annotator class with the process() method that does the work. For aggregates, this can be an (optional) custom flow controller class that computes the flow.

The classes for annotators and flow controllers do not need to be "thread-safe" with respect to their instance data - meaning, they do not need to be implemented with synchronization locks for access to their instance data, because each instance will only be called using one thread at a time. Scale out for these classes is done using multiple instances of the class.

However, if you have class "static" fields shared by all instances, or other kinds of external data shared by all instances (such as a writable file), you must be aware of the possibility of multiple threads accessing these fields or external resources, running on separate instances of the class, and do any required synchronization for these.

1.4.2. AS Component wrapping

Components managed by AS

1. have an associated input queue (this may be internal, or explicit and externalized).

   They receive work units (CASes) from this queue, and return the updated CASes to an output queue which is specified as part of the message delivering the input work unit (CAS).

2. have a container which wraps the component and provides the following services (see Figure 1.1, “AS Primitive Wrapper” [5]):
   - A connection to an input queue of CASes to be processed
   - Scale-out within the JVM for components at the bottom level - the AS Primitives. Scaleout creates multiple instances of the annotator(s), and runs each one on its own thread, all drawing work from the same input queue.
   - (For AS Aggregates) connections to input queues of the delegates
   - A "pull" mechanism for the component to pull new CASes (to be processed) from their associated input queue
   - (For AS Aggregates) A separate, built-in internal queue to receive CASes back from delegates. These are passed to the aggregate's flow controller, which then specifies where they go next.
   - A connection to user-specified error handlers. Error conditions are communicated to the flow controller, to enable user / dynamically determined recovery or termination actions.

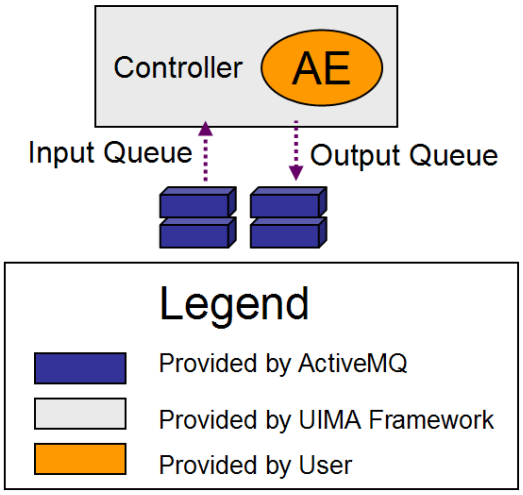

Figure 1.1. AS Primitive Wrapper

As shown in the next figure, when the component being wrapped is an AS Aggregate, the container will use the aggregate's flow controller (shown as "FC") to determine the flow of the CASes among the delegates. The next figure shows the additional output queue configured for aggregates to receive CASes returning from delegates. The dashed lines show how the queues are associated with the components.

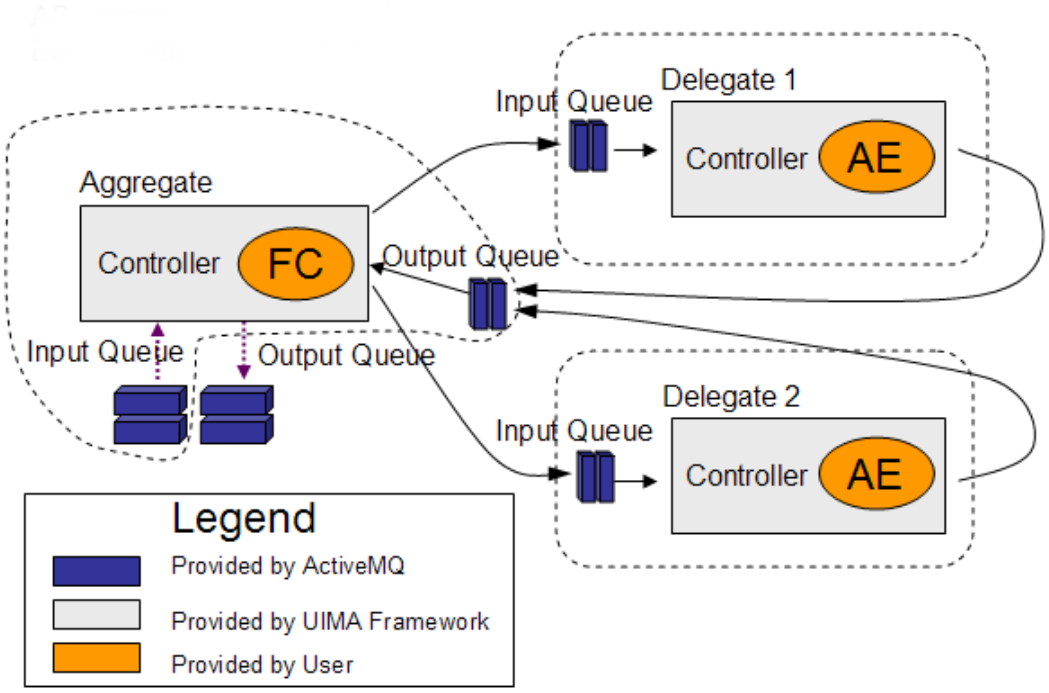

Figure 1.2. AS Aggregate wrapper

The collection of parts and queues is wired together according to a deployment specification, provided by the deployer. This specification is a collection of one or more deployment descriptors.

1.4.3. Parallel Flows

A Flow Controller Parallel Step will actually run in parallel, when using remote delegates with xmi serialization (the default) sepecified. For colocated delegates, or for remote delegates running with binary serialization, the parallel steps will be run serially in some arbitrary unspecified order. For the parts running in parallel on remotes, existing Feature Structures may not be modified - only new ones can be added. This is checked for, when the results from the remote parallel steps are merged.

1.4.4. Deployment alternatives

Deployment is concerned with the following kinds of parts, and allocating these parts (possibly replicated) to various hosts:

- Application Drivers. These represent the top level caller of UIMA functionality. Examples include: stand-alone Java applications, such as the example document analyzer tool, a custom Web servlet, etc.
- AS Services. AS primitive or AS aggregate services deployed on one or more nodes as needed to meet scalability requirements.
- Queue Brokers. Each Queue Broker manages and provides the storage facility for one or more named queues.

Parts can be co-located or not; when they're not, we say they're remote. Remote includes running on the same host, but in a different process space, using a different JVM or other native process. Connections between the non-co-located parts are done using the JMS (Java Messaging Service) protocols, using ActiveMQ from apache.org.

> **Note:** For high availability, the Queue Brokers can be, themselves, replicated over many hosts, with fail-over capability provided by the underlying ActiveMQ implementation.

1.4.4.1. Configuring multiple instances of components

AS components can be replicated; the replicated components can be co-located or distributed across different nodes. The purpose of the replication is to allow multiple work units (CASes) to be processed in parallel, in multiple threads, either in the same host, or using different hosts. The vision is that the deployment is able to replicate just those components which are the bottleneck in overall system thruput.

There are two ways replication can be specified.

1. In the deployment descriptor, for an AS Primitive component, set the numberOfInstances attribute of the <scaleout> element to a number bigger than one.

2. Deploy the same service on many nodes, specifying the same input service queue

The first way is limited to replicating an AS Primitive. An AS Primitive can be the whole component of the service, or it can be at the bottom of an aggregate hierarchy of co-located parts.

Replicating an AS Primitive has the effect of replicating all of its nested components (if it is an aggregate), since no queues are used below its input queue.

1.4.4.2. Queues

Asynchronous operation uses queues to connect components. For co-located components, the UIMA AS framework uses custom very-lightweight queuing mechanisms. For non-co-located

components, it uses JMS queues, managed by ActiveMQ Queue Brokers, which can be running on the other nodes in a network.

AS Aggregate delegates specified as <analysisEngine> elements (or by default) are co-located, and use custom lightweight queuing. AS Aggregate delegates specified using <remoteAnalysisEngine> are not co-located, and use JMS queuing.

For JMS queues, each queue is defined by a queue name and the URL of its Queue Broker. AS services register as queue consumers to obtain CASes to work on (as input) and to send CASes they're finished with (as output) to a reply queue connected to the AS client.

The queue implementation for JMS is provided by ActiveMQ queue broker. A single Queue Broker can manage multiple queues. By default UIMA AS configures the Queue Broker to use in-memory queues; the queue is resident on the same JVM as its managing Queue Broker. ActiveMQ offers several failsafe options, including the use of disk-based queues and redundant master/slave broker configurations.

The decisions about where to deploy Queue Brokers are deployment decisions, made based on issues such as domain of control, firewalls, CPU / memory resources, etc. Of particular interest for distributed applications is that a UIMA AS service can be deployed behind a firewall but still be publicly available by using a queue broker that is available publicly.

When components are co-located, an optimization is done so that CASes are not actually sent as they would be over the network; rather, a reference to the in-memory Java object is passed using the queue.

> **Warning:** Do not hook up different kinds of services to the same input queue. The framework expects that multiple services all listening to a particular input queue are sharing the workload of processing CASes sent to that queue. The framework does not currently verify that all services on a queue are the same kind, but likely will in a future release.

1.4.4.3. Deployment Descriptors

Each deployment descriptor specifies deployment information for one service, including all of its co-located delegates (if any). A service is an AS component, having one top level input queue, to which CASes are sent for processing.

Each deployment descriptor has a reference to an associated Analysis Engine descriptor, which can be an aggregate, or a primitive (including CAS Consumers).

AS Components can be co-located (this is the default); the deployment descriptor specifies remote queues (queue-brokers and queue-names) for non-co-located components.

All services need to be manually started using an appropriate deployment descriptor (describing the things to be set up on that server). There are several scripts provided including deployAsyncService, that do this. The client API also supports a deploy method for doing this within the same JVM.

Deploying UIMA aggregates

UIMA aggregates can either be run asynchronously as AS Aggregates, or synchronously (as AS Primitives). AS Aggregates have an input and a reply queue associated with each delegate, and can process multiple CASes at a time. UIMA aggregates that are run as AS Primitives send CASes synchronously, one a time, to each delegate, without using any queuing mechanism.

Each delegate in an AS Aggregate can be specified to be local or remote. Local means co-located using internal queues; remote means all others, including delegates running in a different JVM, or in the same JVM but that can be shared by multiple clients. For each delegate which is remote, the deployment descriptor specifies the delegate's input queue; a corresponding reply queue is also automatically set up. If the delegate is local, internal input and reply queues are automatically created for that delegate.

1.4.5. Current design limitations

This section describes limitations of the current support for AS.

1.4.5.1. Sofa Mapping limits

Sofa mapping works for co-located delegates, only. As with Vinci and SOAP, remote delegates needing sofa mapping need to respecify sofa mappings in an aggregate descriptor at the remote node.

1.4.5.2. Parameter Overriding limits

Parameter overrides only work for co-located delegates. As with Vinci and SOAP, remote delegates needing parameter overrides need to respecify the overrides in an aggregate descriptor at the remote node.

1.4.5.3. Resource Sharing limits

Resource Sharing works for co-located delegates, only.

1.4.6. Compatibility with earlier version of remoting and scaleout

There is a new type of client service descriptor for an AS service, the JMS service descriptor (see Section 1.7, “JMS Service Descriptor” [9]), which can be used along with Vinci and/or SOAP services in base UIMA applications. Conversely, Vinci services **cannot** be used within a UIMA AS service because they do not comply to the UIMA standard requiring preservation of feature structure IDs. SOAP service calls currently use a binary serialization of the CAS which does preserve IDs and therefore can be called from a UIMA AS service.

To use SOAP services within a UIMA AS deployment, wrap them inside another aggregate (which might contain just the one SOAP service descriptor), where the wrapping aggregate is deployed as an AS Primitive.

1.5. Application Concepts

When UIMA is used, it is called using Application APIs. A typical top-level driver has this basic flow:

1. Read UIMA descriptors and instantiate components
2. Do a Run
3. Do another Run, etc.
4. Stop

A "run", in turn, consists of 3 parts:

1. initialize (or reinitialize, if already run)
2. process CASes
3. finish (collectionProcessComplete is called)

Initialize is called by the framework when the instance is created. The other methods need to be called by the driver. `collectionProcessComplete` should be called when the driver determines that it is finished sending input CASes for processing using the `process()` method. `reinitialize()` can be called if needed, after changing parameter settings, to get the co-located components to reinitialize.

1.5.1. Application API

See Chapter 4, *Asynchronous Scaleout Application Interface* [27] and the sample code.

1.5.2. Collection Process Complete

An application may want to signal a chain of annotators that are being used in a particular "run" when all CASes for this run have been processed, and any final computation and outputting is to be done; it calls the collectionProcessComplete method to do this. This is frequently done when using stateful components which are accumulating information over multiple documents.

It is up to the application to determine when the run is finished and there are no more CASes to process. It then calls this method on the top level analysis engine; the framework propagates this method call to all delegates of this aggregate, and this is repeated recursively for all delegate aggregates.

This call is synchronous; the framework will block the thread issuing the call until all processing of CASes within the service has completed and the collectionProcessComplete method has returned (or timed out) from every component it was sent to.

Components receive this call in a fixed order taken from the <fixedFlow> sequence information in the descriptors, if that is available, and in an arbitrary order otherwise.

If a component is replicated, only one of the instances will receive the collectionProcessComplete call.

1.6. Monitoring and Controlling an AS application

JMX (Java Management Extensions) are used for monitoring and controlling an AS application. As of release 2.3.0, extensive monitoring facilities have been implemented; these are described in a separate chapter on Chapter 5, *Monitoring, Tuning and Debugging* [33]. The only controlling facility provided is to stop a service.

In addition, a configurable Monitoring program is provided which works with the JMX provided measurements and aggregates and samples these over specified intervals, and creates monitoring entries in the UIMA log, for tuning purposes. You can use this to detect overloaded and/or idle services; see the Chapter 5, *Monitoring, Tuning and Debugging* [33] chapter for details.

1.7. JMS Service Descriptor

To call a UIMA AS Service from Document Analyzer or any other base UIMA application, use a descriptor such as the following:

```
<customResourceSpecifier xmlns="http://uima.apache.org/resourceSpecifier">
  <resourceClassName>
    org.apache.uima.aae.jms_adapter.JmsAnalysisEngineServiceAdapter
  </resourceClassName>
  <parameters>
    <parameter name="brokerURL"
               value="tcp://uima17.watson.ibm.com:61616"/>
    <parameter name="endpoint"
               value="uima.as.RoomDateMeetingDetectorAggregateQueue"/>
    <parameter name="timeout"
               value="xxx"/>
    <parameter name="getmetatimeout"
               value="yyy"/>
  </parameters>
</customResourceSpecifier>
```

The resourceClassName must be set exactly as shown. Set the brokerURL and endpoint parameters to the appropriate values for the UIMA AS Service you want to call. These are the same settings you would use in a deployment descriptor to specify the location of a remote delegate. Note that this is a synchronous adapter, which processes one CAS at a time, so it will not take advantage of the scalability that UIMA AS provides. To process more than one CAS at a time, you must use the Asynchronous UIMA AS Client API Chapter 4, *Asynchronous Scaleout Application Interface* [27].

Other parameters may be specified:

binary_serialization
: Set to true to specify binary serialization (faster, but requires that the service have exactly the same type system as the client).

ignore_process_errors
: Set to true to specify that any processing errors should be ignored. In order for this to be reasonable, your calling environment must be able to continue somehow, if the service fails.

For more information on the customResourceSpecifier see Section 2.8, “Custom Resource Specifiers”.

1.8. Life cycle

Running UIMA AS applications involves deploying (starting) UIMA AS services, perhaps over a wide area network, perhaps on many machines. UIMA AS as a few preliminary tools to help. These include the ability of the Chapter 4, *Asynchronous Scaleout Application Interface* [27] to deploy UIMA AS services (limited to deployment within the same JVM), and scripts such as `deployAsyncService` that start up a UIMA AS Service.

`deployAsyncService` has a facility that launches a keyboard listener after starting, which listens for a "s" or "q" keystroke. The "s" stops the service immediately, and the "q" quiesces the service, letting any in-process work finish before stopping.

JMX beans for services include a control option to stop the service.

Chapter 2. Error Handling for Asynchronous Scaleout

This chapter discusses the high level architecture for Error Handling from the user's point of view.

2.1. Basic concepts

This chapter describes error configuration for AS components.

The AS framework manages a collection of component parts written by users (user code) which can throw exceptions. In addition, the AS framework can run timers when sending commands to user code which can create timeouts.

An AS component is either an AS aggregate or an AS primitive. AS aggregates can have multiple levels of aggregation; error configuration is done for each level of aggregation. The rest of this chapter focuses on the error configuration one level at a time (either for one particular level in an aggregate hierarchy, or for an AS primitive).

There is a small number of commands which can be sent to an AS component. When a component returns the result, if an error occurs, an error result is returned instead of the normal result.

Configuration and support is provided for three classes of errors:

1. Exceptions thrown from code (component or framework) at this level
2. error messages received from delegates.
3. timeouts of commands sent to delegates.

The second and third class of errors is only possible for AS aggregates.

When errors happen, the framework provides a standard set of configurable actions. See Section 2.8, “Commands and allowed actions” [16] for a summary table of the actions available in different situations.

2.2. Associating Errors with incoming commands

Components managed by AS may generate errors when they are sent a command. The error is associated with the command that was running to produce the error.

There are three incoming message commands supported by the AS framework:

1. getMetadata - sent by the framework when initializing connection to an AS component
2. processCas - sent once for each CAS
3. collectionProcessComplete - sent when an application calls this method

Error handling actions are associated with these various commands. Some error handling actions make sense only if there is an associated CAS object, and are therefore only allowed with the processCas command.

2.2.1. Error handling for CASes generated in an Aggregate by CAS Multipliers

CASes that are generated by a CAS Multiplier are called child CASes, and their parent CAS is the CAS that originally came into the CAS Multiplier which caused the child CASes to be created. Each child CAS always has one associated parent CAS.

The flow of CASes is constrained to always block returning the parent CAS until all of its children have been generated by the CAS Multiplier. In addition, the framework (currently) blocks the flow of the parent CAS until all of its children have finished all processing, including any processing of the children in outer, containing aggregates (which can even be on other network-connected nodes). (There is some discussion about relaxing this condition, to allow more asynchronicity.)

A child CAS may only exist for a part of the flow, and not returned all the way back up to the top. Because of this, errors which occur on a child CAS and are not recovered are reported on both the child CAS, and on its parent. The parent CAS is not processed further, and an error is reported against the parent.

The parent CAS may have other outstanding children currently being processed. It is not yet specified what happens (if anything) to these CASes.

2.3. Error handling overview

When an error happens, it is either "recovered", or not; only errors from delegates of an AS aggregate can be recovered. Recovery may be achieved by retrying the request or by skipping the delegate.

Commands normally return results; however if an non-recoverable error occurs, the command returns an error result instead.

For AS aggregates, each level in aggregate hierarchy can be configured to try to recover the error. If a particular AS aggregate level does not recover, the error is sent up to the next level of the hierarchy (or to the calling client, if a top level). The error result is updated to reflect the actions the framework takes for this error.

Non-recovered errors can optionally have an associated "Terminate" or "Disable" action (see below), triggered by the error when a threshold is reached. "Disable" applies to the delegate that generated the error while "Terminate" applies to the aggregate and any co-located aggregates it is contained within.

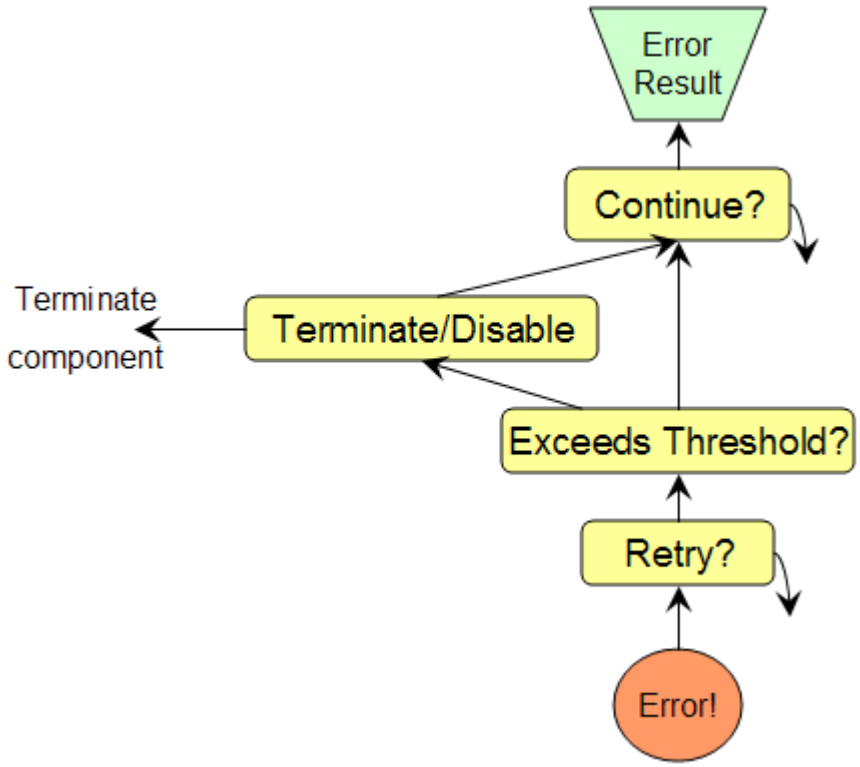

Figure 2.1. Basic error handling chain for AS Aggregates for errors from delegates

The basic error handling chain starts with an error, and can attempt to recover using retry. If this fails (or is not configured), the error count is incremented and checked against the thresholds for this delegate. If the threshold has been reached the specified action is taken, disabling the delegate or terminating the entire component. If the Terminate error is not taken, recovery by the Continue action can be attempted. If that fails, an error result is returned to the caller.

For AS primitives, only the Terminate action is available, and Retry and Continue do not apply.

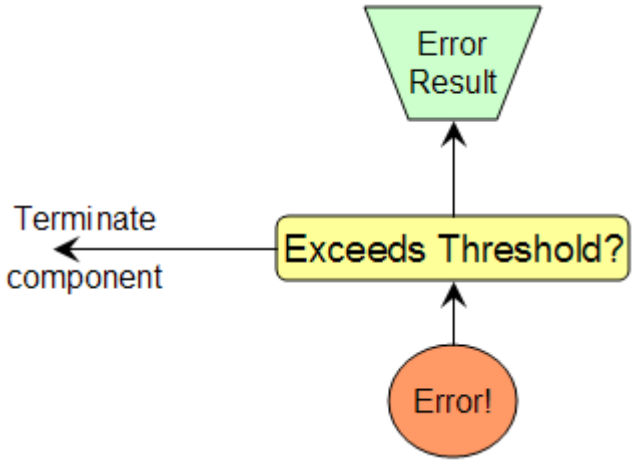

Figure 2.2. Basic error handling chain for AS Primitives

2.4. Error results

Error results are returned instead of a CAS, if an error occurs and is not recovered.

If the application uses the synchronous sendAndReceive() method, an Error Result is passed back to the client API in the form of a Java exception. The particular exception varies, depending on many factors, but will include a complete stack trace beginning with the cause of the error. If the application uses an asynchronous API, the exception is wrapped in a EntityProcessStatus object and delivered via a callback listener registered by the application. See section 4.4 Status Callback Listener for details.

2.5. Error Recovery actions

When errors occur in delegates, the aggregate containing them can attempt to recover. There are two basic recovery actions: retrying the same command and continuing past (skipping) the failing component.

Every command sent to a delegate can have an associated (configurable) timeout. If the timeout occurs before the delegate responds, the framework creates an error representing the timeout.

> **Note:** If, subsequently, a response is (finally) received corresponding to the command that had timed-out, this is logged, but the response is discarded and no further action is taken.

When errors occur in delegates, retry is attempted (if configured), some number of times. If that fails, error counts are incremented and thresholds examined for Terminate/Disable actions. If not reached, or if the action is Disable, Continue is attempted (if configured); if Continue fails, the error is not recovered, and the aggregate returns the error result from the delegate to the aggregate's caller. On Terminate, the error is returned to the caller.

2.5.1. Aggregate Error Actions

This section describes in more detail the error actions valid only for AS aggregates.

2.5.1.1. Retry

Retry is an action which re-sends the same command that failed back to the input queue of the delegate. (Note: It may be picked up by a different instance of that delegate, which may have a better chance of succeeding.) The framework will keep a copy of the CAS sent to remote components in order to have it to send again if a retry is required.

Retry is not allowed for colocated delegates.

The "collectionProcessComplete" command is never retried.

Retry is done some number of times, as specified in the deployment descriptor.

2.5.1.2. Disable Action

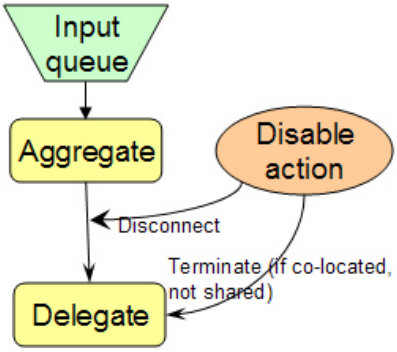

Figure 2.3. Disable action

When this action is taken the framework marks the particular delegate causing the error as "disabled" so it will be skipped in subsequent calls. The framework calls the flow controller, telling it to remove the particular delegate from the flow.

2.5.1.3. Continue Option on Delegate Process CAS Failures

For processCas commands, the Continue action causes the framework to give the flow controller object for that CAS the error details, and ask the flow controller if processing can continue. If the flow controller returns "true", the flow controller will be called asking for the next step; if "false", the framework stops processing the CAS, returning an error to the client reply queue, or just returning the CAS to the casPool of the CAS multiplier which created it.

For "collectionProcessComplete" commands, Continue means to ignore the error, and continue as if the collectionProcessComplete call had returned normally.

This action is not allowed for the getMetadata command.

2.6. Thresholds for Terminate and Disable

The Terminate and Disable actions are conditioned by testing against a threshold.

Thresholds are specified as two numbers - an error count and a window. The threshold is reached if the number of errors occurring within the window size is equal to or greater than "the error count". A value of 0 disables the error threshold so no action can be taken. A window of 0 means no window, i.e. all errors are counted

Errors associated with the processCas command are the only ones that are counted in the threshold measurements.

2.7. Terminate Action

When this action is taken the service represented by this component sends an error reply and then terminates, disconnecting itself as a listener from its input queue, and cleaning itself up (releasing resources, etc.). During cleanup, the component analysis engine's `destroy` method is called.

> **Note:** The termination action applies to the entire aggregate service. Remote delegates are not affected.

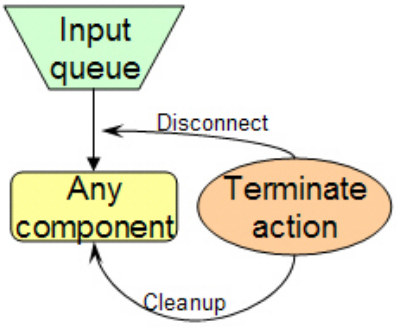

Figure 2.4. Terminate action

If the threshold is not exceeded, the error counts associated with the threshold are incremented.

> **Note:** Some errors will always cause a terminate: for instance, framework or flow controller errors cause immediate termination.

2.8. Commands and allowed actions

All of the allowed actions are optional, and default to not being done, except for getMetadata being sent to a delegate that is remote - this has a default timeout of 1 minute.

Here's a table of the allowed actions, by command. In this table, the Retry, Continue, and Disable actions apply to the particular Delegate associated with the error; the Terminate action applies to the entire component.

The framework returns an Error Result to the caller for errors that are not recovered.

Table 2.1. Error actions by command type

Command	**Error actions allowed**	
	AS Aggregate	**AS Primitive**
getMetadata	Retry, Disable, Terminate	Terminate
processCas	Retry, Continue, Disable, Terminate	Terminate
collection Processing Complete	Continue, Disable, Terminate	Terminate

The rationale for providing the terminate action for primitive services is that only the service can know that it is no longer capable of continued operation. Consider a scaled component with multiple service instances, where one of them goes "bad" and starts throwing exceptions: the clients of this service have no way of stopping new requests from being delivered to this bad service instance. The teminate action allows the bad service to remove itself from further processing; this could allow life cycle management to restart a new instance.

Chapter 3. Asynchronous Scaleout Deployment Descriptor

3.1. Descriptor Organization

Each deployment descriptor describes one service, associated with a single UIMA descriptor (aggregate or primitive), and describes the deployment of those UIMA components that are co-located, together with specifications of connections to those subcomponents that are remote.

The deployment descriptor is used to augment information contained in an analysis engine descriptor. It adds information concerning

- which components are managed using AS
- queue names for connecting components
- error thresholds and recovery / terminate action specifications
- error handling routine specifications

The application can include both Java and non-Java components; the deployment descriptors are slightly different for non-Java components.

3.2. Deployment Descriptor

Each deployment descriptor describes components associated with one UIMA descriptor. The basic structure of a Deployment Descriptor is as follows:

```
<analysisEngineDeploymentDescription
      xmlns="http://uima.apache.org/resourceSpecifier">

  <!-- the standard (optional) header -->
  <name>[String]</name>
  <description>[String]</description>
  <version>[String]</version>
  <vendor>[String]</vendor>

  <deployment protocol="jms" provider="activemq">

    <casPool numberOfCASes="xxx" initialFsHeapSize="nnn"/>

    <service>         <!-- must have only 1 -->

      <!-- 0 or more of the following -->
      <!-- name required, value optional -->
      <custom name="..." value="..."/>

      <inputQueue .../>

      <topDescriptor .../>

      <environmentVariables .../>  <!--optional -->

      <analysisEngine key="key name" async="[true/false]"
             internalReplyQueueScaleout="nn1" <!-- optional -->
             inputQueueScaleout="nn2">        <!-- optional -->
```

```
        <scaleout numberOfInstances="1"/>       <!-- optional -->
                                                <!-- optional -->
        <casMultiplier
             poolSize="5"                       <!-- optional -->
             initialFsHeapSize="nnn"            <!-- optional -->
             processParentLast="[true/false]"/> <!-- optional -->

        <asyncPrimitiveErrorConfiguration .../> <!-- optional -->

        <delegates>    <!-- optional, only for aggregates -->
                                    <!-- 0 or more -->
          <analysisEngine key="key name" async="[true/false]"
               internalReplyQueueScaleout="nn1"
               inputQueueScaleout="nn2">

                ...    <!-- optional nested specifications -->
          </analysisEngine>
                . . .
          <remoteAnalysisEngine key="key name"  <!-- 0 or more -->
               remoteReplyQueueScaleout="nn1">  <!-- optional -->

            <!-- next is either required or must be omitted -->
            <casMultiplier
                    poolSize="5"
                    initialFsHeapSize="nnn"
                    processParentLast="[true/false]"/> <!-- optional -->

            <inputQueue ... />
            <serializer method="xmi"/>
            <asyncAggregateErrorConfiguration ... />
          </remoteAnalysisEngine>
                . . .
        </delegates>
      </analysisEngine>
    </service>
  </deployment>
</analysisEngineDeploymentDescription>
```

3.3. CAS Pool

This element specifies information for managing CAS pools. Having more CASes in the pools enables more AS components to run at the same time. For instance, if your application had four components, but one was slow, you might deploy 10 instances of the slow component. To get all 10 instances working on CASes simultaneously, your CAS pool should be at least 10 CASes. The casPool size should be small enough to avoid paging.

The initialFsHeapSize attribute is optional, and allows setting the size of the initial CAS Feature Structure heap. This number is specified in bytes, and the default is approximately 2 megabytes for Java top-level services, and 40 kilobytes for C++ top level services. The heap grows as needed; this parameter is useful for those cases where the expected heap size is much smaller than the default.

3.4. Service

This section is required and specifies the deployment information for the service.

3.5. Customizing the deployment

The <custom> element(s) are optional. Each one, if specified, requires a name parameter, and can have an optional value parameter. They are intended to provide additional information needed for particular kinds of deployment.

The following lists the things that can be specified here.

- name="run_top_level_CPP_service_as_separate_process"

  (no value used)

  Causes the top level component, which must be a component specified as using <frameworkImplementation>org.apache.uima.cpp</frameworkImplementation> and which must be specified as async="false" (the default), to be run in a separate process, rather than via using the JNI.

3.6. Input Queue

The inputQueue element is required. It identifies the input queue for the service.

```
<inputQueue brokerURL="tcp://x.y.z:portnumber"
    endpoint="queue_name"
    prefetch="1"/>
```

The brokerURL attribute is optional. When omitted, a default value of tcp://localhost:61616 will be used. A different brokerURL can be provided as an override when launching a service. Consult README that provides an example of brokerURL override. The queue broker address includes a protocol specification, which should be set to either "tcp", or "http". The brokerURL attribute specifies the queue broker URL, typically its network address and port. .

The http protocol is similar to the tcp protocol, but is preferred for wide-area-network connections where there may be firewall issues, as it supports http tunnelling.

> **Warning:** When remote delegates are being used, the brokerURL value used for this remote delegate is used also for the remote reply Queue, and must be valid for both the client to send requests and the remote service to send replies to. The URL to use for the reply is resolved on the remote system when sending a reply. Using "localhost" will not work, nor will partially specified URLs unless they resolve to the same URL on all nodes where services are running. The recommended best practice is to use fully qualified URL names.

The queue name is used to uniquely identify a queue belonging to a particular broker.

The `prefetch` attribute controls prefetching of messages for an instance of the service. It can be 0 - which disables prefetching. This is useful in some realtime applications for reducing latency. In this case, when a new request arrives, any available instance will take the request; if prefetching was set above 0, the request might be prefetched by a busy service. The default value if not specified is 0.

> **Note:** The `prefetch` attribute is only used with the top inputQueue element for the service.

3.7. Top level Analysis Engine descriptor

Each service must indicate some analysis engine to run, using this element.

```
<topDescriptor>
  <import location="..." /> <!-- or name="..." -->
</topDescriptor>
```

This is the standard UIMA import element. Imports can be by name or by location; see Section 2.2, “Imports”.

3.8. Setting Environment Variables

This element is optional, and provides a way to set environment variables.

> **Note:** This element is only allowed and used for top level Analysis Engines specifying <frameworkImplementation>org.apache.uima.cpp</frameworkImplementation> and running using the <custom name="run_top_level_CPP_service_as_separate_process">; it is not supported for Java Analysis Engines.

Components written in C++ can be run as a top level service. These components are launched in a separate process, and by default, all the environment variables of the launching process are passed to the new process. This element allows the environment variables of the new process to be augmented.

```
<environmentVariables>
<!-- one or more of the following element -->
<environmentVariable name="xxx">value goes here</environmentVariable>
</environmentVariables>
```

Usually, the value will replace any existing value. As a special exception, for the environment variables used as the PATH (for Windows) or LD_LIBRARY_PATH (for Linux) or DYLD_LIBRARY_PATH (for MacOS), the value will be "prepended" with a path separator character appropriate for the platform, to any existing value.

3.9. Analysis Engine

This is used to describe an element which is an analysis engine. It is optional and only needed if the defaults are being overridden. The `async` attribute is only used for aggregates, and specifies that this aggregate will be run asynchronously (with input queues in front of all of its delegates) or not. If not specified, the async property defaults to "false" except in the case where the deployment descriptor includes the <delegates> element, when it defaults to "true". If you specify async="false", then it is an error to specify any <delegates> in the deployment descriptor.

The `key` attribute must have as its value the key name used in the containing aggregate descriptor to uniquely identify this delegate. Since the top level aggregate is not contained in another aggregate, this can be omitted for that element. Deployment information is matched to delegates using the key name specified in the aggregate descriptor to identify the delegate.

```
<analysisEngine key="key name" async="true"
             internalReplyQueueScaleout="nn1"  <!-- optional -->
             inputQueueScaleout="nn2">         <!-- optional -->
```

```
  <scaleout numberOfInstances="1"/>        <!-- optional  -->
  <!-- casMultiplier is either required, or must be omitted-->
  <casMultiplier
       poolSize="5"                        <!-- optional  -->
       initialFsHeapSize="nn"              <!-- optional  -->
       processParentLast="[true/false]"/>  <!-- optional  -->

    <!-- next two are optional, but only one allowed -->
  <asyncAggregateErrorConfiguration .../>  <!-- optional  -->
  <asyncPrimitiveErrorConfiguration .../>  <!-- optional  -->

  <delegates>                              <!-- optional  -->
    <analysisEngine key="key name" ...>    <!-- 0 or more -->
            ...       <!-- optional nested specifications -->
    </analysisEngine>
            . . .
    <remoteAnalysisEngine key="key name"   <!-- 0 or more -->
             remoteReplyQueueScaleout="nn1"> <!-- optional -->

      <!-- next is either required or must be omitted -->
      <casMultiplier
           poolSize="5"                        <!-- optional  -->
           initialFsHeapSize="nnn"             <!-- optional  -->
           processParentLast="[true/false]"/>  <!-- optional  -->

      <inputQueue ... />
      <serializer method="[xmi|binary]"/>             <!-- optional -->
      <asyncAggregateErrorConfiguration .../> <!-- optional -->
    </remoteAnalysisEngine>
            . . .
  </delegates>               . . .
</analysisEngine>
```

<analysisEngine> is used to specify deployment details for an analysis engine. It is optional, and if omitted, defaults will be used: The analysis engine will be run synchronously (processing only one CAS at a time), with a scaleout of 1, using the default error configuration.

The attributes `internalReplyQueueScaleout` and `inputQueueScaleout` only have meaning and are allowed when async="true" is specified (which in turn can only be set true for aggregates) or is the default (which happens when the aggregate has delegate deployment options specified in the deployment descriptor). These attributes default to 1. For asynchronous aggregates, they control the number of threads used to do the work of the aggregate outside of running the delegates. This work can include one or more of the following:

- deserializing an input CAS (only on the input Queue), or serializing the resulting CAS back to a remote requester (only if the requester is remote).
- running the flow controller
- serializing CASes being sent to remote delegates (only useful if one or more of the delegates is remote).

These attributes provide a way to scale out this work on multi-core machines, if these tasks become a bottleneck.

Note that if an aggregates flow controller specifies that the first delegate the CAS should flow to is a remote, the work of serializing the CAS to that remote is done using the inputQueue thread, and the scaleout parameter that would apply would be the inputQueueScaleout. For subsequent delegates, the work is done on the internalReplyQueueScaleout threads.

The <scaleout ...> element specifies, for co-located primitive or non-AS aggregates (async="false") at the bottom of an aggregate tree, how many replicated instances are created.

The <casMultiplier> element inside an <analysisEngine> element is required if the analysis engine component is a CAS multiplier, and is an error if specified for other components. It specifies for CAS multipliers the size of the pool of CASes used by that CAS multiplier for generating extra CASes.

> **Note:** The actual CAS pool size can be bigger than the size specified here. The custom CAS multiplier code specifies how many CASes it needs access to at the same time; the actual CAS pool size is the value in the deployment descriptor, plus the value in custom CM code, minus 1.

The initialFsHeapSize attribute on the <casMultiplier> element is optional, and allows setting the size of the initial CAS Feature Structure heap for CASes in this pool. This number is specified in bytes, and the default is approximately 2 megabytes for Java top-level services, and 40 kilobytes for C++ top level services. The heap grows as needed; this parameter is useful for those cases where the expected heap size is much smaller than the default.

The processParentLast attribute on the <casMultiplier> element is optional, and specifies processing order of an input CAS relative to its children. If true, a flow of an input CAS will be suspended after it is returned from a Cas Multiplier delegate until all its child CASes have finished processing. If false, an input CAS can be processed in parallel with its children.

The <remoteAnalysisEngine> elements are used to specify that the delegate is not co-located, and how to connect to it. The `remoteReplyQueueScaleout` is optional; if not specified it defaults to 1. This scaleout is the number of threads that will be used to do the work of the containing aggregate when replies are returned from this remote delegate. This work is described above. It may be useful to set this to > 1 if, for instance, there are many CASes coming back from a remote delegate (perhaps the remote is a CAS Multiplier), and each one has to be deserialized.

The <serializer> element describes what method of serialization to use. This element is optional and it may be set to either `binary` or `xmi`. If omitted, `xmi` serialization will be used by default. `Xmi` serialization can be quite verbose and produce large output for CASes containing many annotations; on the plus side, it supports serialization between components where the type systems may not be exactly identical (for instance, they could be different subsets of larger, common type systems). `Binary` serialization produces a smaller output size and is more efficient; on the minus side, it requires that the type systems for both components have exactly the same type and feature codes - which in practice means that the type systems have to be identical. Also, the binary serialization format is new with 2.3.0 release, and is not always available. For example, C++ services do not (currently) support this format.

The <inputQueue> element specifies the remote's input queue. The casMultiplier element inside a remoteAnalysisEngine element is only specified if the remote component is a CAS Multiplier, and it specifies the size of a pool of CASes kept to receive the new CASes from the remote component, and the initial size of those CASes. Its poolSize must be equal to or larger than the casMultiplier poolSize specified for that remote component.

> **Note:** As of release 2.3.1, the previous restrictions limiting remote CAS Multiplier to just one have been lifted; you can have any number, and they can be scaled out as well.

> **Note:** The brokerURL value used for this remote delegate must be valid for both the client to send requests and the remote service to send replies.

Services may be running on nodes with firewalls, where the only port open is the one for http. In this case, you can use the http protocol.

The <asyncPrimitiveErrorConfiguration> element is only allowed within a top-level analysis engine specification (that is, one that is not a delegate of another, containing analysis engine).

3.10. Error Configuration descriptors

Error Configuration descriptors can be included directly in the deployment descriptors, or they may use the <import> mechanism to import another file having the specification.

For AS Aggregates, the configuration applicable to delegates goes in <asyncAggregateErrorConfiguration> elements for the delegate.

For AS Primitives, there is one <asyncPrimitiveErrorConfiguration> element that configures threshold-based termination. The other kinds of error configuration are not applicable for AS Primitives.

See ???? for a complete overview of error handling.

The Error Configuration descriptor for AS Aggregates is as follows; note that all the elements are optional:

```
<asyncAggregateErrorConfiguration
      xmlns="http://uima.apache.org/resourceSpecifier">

  <!-- the standard (optional) header -->
  <name>[String]</name>
  <description>[String]</description>
  <version>[String]</version>
  <vendor>[String]</vendor>

  <import ... />  <!-- optional -->

  <getMetadataErrors
          maxRetries="n"
          timeout="xxx_milliseconds"
          errorAction="disable|terminate"/>

  <processCasErrors
          maxRetries="n"
          timeout="xxx_milliseconds"
          continueOnRetryFailure="true|false"
          thresholdCount="xxx"
          thresholdWindow="yyy"
          thresholdAction="disable|terminate"/>

  <collectionProcessCompleteErrors
          timeout="xxx_milliseconds"
          additionalErrorAction="disable|terminate"/>

</asyncAggregateErrorConfiguration>
```

For an AS Primitive, the <asyncPrimitiveErrorConfiguration> element appears at the top level, and has this form:

```
<asyncPrimitiveErrorConfiguration
      xmlns="http://uima.apache.org/resourceSpecifier">

  <!-- the standard (optional) header -->
  <name>[String]</name>
```

```
  <description>[String]</description>
  <version>[String]</version>
  <vendor>[String]</vendor>

  <import ... />  <!-- optional -->

  <processCasErrors
          thresholdCount="xxx"
          thresholdWindow="yyy"
          thresholdAction="terminate"/>

  <collectionProcessCompleteErrors
          additionalErrorAction="terminate"/>

</asyncPrimitiveErrorConfiguration>
```

The maxRetries attribute specifies the maximum number of retries to do. If this is set to 0 (the default), no retries are done.

The continueOnRetryFailure attribute, if set to 'true' causes the framework to ask the aggregate's flow controller if the processing for the CAS can continue. If this attribute is 'false' or if the flow controller indicates it cannot continue, further processing on the CAS is stopped and an error is returned from the aggregate. Warning: there are some conditions in the current implementation where this is not yet being done; this is a known issue.

> **Warning:** If maxRetries > 0 or the continueOnRetryFailure attribute is 'true', the CAS will be saved before sending it to remote delegates, to enable the these actions. For co-located delegates, the CAS is *not* copied, therefore the retry and continue options are not allowed.

The timeout attribute specifies the timeout values used when sending commands to the delegates. The units are milliseconds and a value of 0 has the special meaning of no timeout.

The thresholdCount and thresholdWindow attributes specify the threshold at which the thresholdAction is taken. If xxx errors occur within a window of size yyy, the framework takes the specified action of either disabling this delegate, or terminating the containing AS Aggregate (or if not an AS Aggregate, terminating the AS Primitive). A thresholdCount of 0 (the default) has the special meaning of no threshold, i.e. errors ignored, and a thresholdWindow of 0 (the default) means no window, i.e. all errors counted.

An action of 'disable' applies to the specified delegate, removing it from the flow so the containing aggregate will no longer send it commands. The 'terminate' action applies to the entire service containing this component, disconnecting it from its input queue and shutting it down. Note that when disabling, the framework asks the flow controller to remove the delegate from the flow, but if the flow controller cannot reasonably operate without this component it can convert the action to 'terminate' by throwing an AnalysisEngineProcessException.FLOW_CANNOT_CONTINUE_AFTER_REMOVE exception.

Note that the only action for an AS Primitive on getMetadata failure is to terminate, and this is always the case, so it is not listed as an configuration option. This is also the default action for an AS Aggregate getMetadata failure.

3.11. Error Configuration defaults

If the <errorConfiguration> element is omitted, or if some sub elements of this are omitted, the following defaults are used:

- The maxRetries parameter is set to 0.
- Timeout defaults are set to 0, meaning no timeout, except for the getMetadata command for remote delegates; here the default is 60000 (1 minute)
- The continueOnRetryFailure action is set to "false".
- The thresholdCount value is set to 0, meaning no threshold, errors are ignored.
- The thresholdWindow value is set to 0, meaning no window, all errors are counted.
- No disable or terminate action will be done (i.e. errors ignored), except for the getMetadata command where the default is to terminate.

Chapter 4. Asynchronous Scaleout Application Interface

4.1. Asynchronous Client API Overview

The Asynchronous Client API provides Java applications the capability to connect to and make requests to UIMA-AS services. ProcessCas and CollectionProcessingComplete requests are supported.

It provides four kinds of capabilities:

- sending requests, receiving replies, asynchronously (or synchronously)
- setting timeouts and limits on the number of simultaneous requests in process (via setting the CAS Pool size)
- using an optionally provided collection reader to obtain items to process
- deploying services as part of the startup process

An application can use this API to prepare and send each CAS to a service one at a time, or alternately can use a UIMA collection reader to prepare the CASes to be delivered.

The application normally provides a listener class to receive asynchronous replies. For individual CAS requests a synchronous sendAndReceive call is available. As an alternative for this synchronous call, instead of using this client API, the standard UIMA Analysis Engine APIs can be used with an analysis engine instantiated from a JMS Service Descriptor. See Section 1.7, “JMS Service Descriptor” [9].

As a convenience, the Asynchronous Client API can also be used to deploy (i.e., "start") services. Java services deployed by the API are instantiated in the same JVM. Logging for all UIMA components in the same JVM are merged; class names and thread IDs can be used to distinguish log entries from different services. All services in the JVM can be monitored by a single JMX console. Native C++ UIMA services can be called from the JVM via the JNI or optionally be launched as separate processes on the same machine. In either case logging and JMX monitoring for native services are integrated with the other UIMA components in the JVM.

4.2. The UimaAsynchronousEngine Interface

An application developer's starting point for accessing UIMA-AS services is the UimaAsynchronousEngine Interface. For each service an application wants to use, it must instantiate a client object:

```
UimaAsynchronousEngine uimaAsEngine =
        new BaseUIMAAsynchronousEngine_impl();
```

The following is a short introduction to some important methods on this class.

- `void initialize(Map anApplicationContext)`: Initializes an asynchronous client. Using configuration provided in the given Map object, this method creates a connection to the UIMA-AS Service queue, creates a response queue, and retrieves

the service metadata. This method blocks until a reply is received from the service or a timeout occurs. If a collection reader has been specified, its typesystem is merged with that from the service. The combined typesystem is used to create a Cas pool. On success the application is notified via the listener's initializationComplete() method, which is called prior to the original call unblocking. Asynchronous errors are delivered to the listener's entityProcessComplete() method. See Section 4.3, “Application Context Map” [29] for more about the ApplicationContext map.

- `void addStatusCallbackListener(UimaASStatusCallbackListener aListener)`: Plugs in an application-specific listener. The application receives callbacks via methods in this listener class. More than one listener can be added.

- `CAS getCAS()`: Requests a new CAS instance from a CAS pool. This method blocks until a free instance of CAS is available in the CAS pool. Applications that use synchronous `sendAndReceive()` and `getCAS()` need to call `CAS.reset()` before reusing the CAS, or `CAS.release()` to return it to the CAS pool. Applications that use asynchronous `sendCAS()` and `getCAS()` must not call `CAS.release()` nor `CAS.reset()` unless `sendCAS()` throws an exception. If `sendCAS()` call is successful, the UIMA AS framework code releases each CAS automatically when a reply is received. The framework releases a CAS right after a callback listener `entityProcessComplete()` completes.

- `void sendCAS(CAS aCAS)`: Sends a given CAS for analysis to the UIMA-AS Service. The application is notified of responses or timeouts via `entityProcessComplete()` method.

- `void setCollectionReader(CollectionReader aCollectionReader)`: Plugs in an instantiated CollectionReader instance to use. This method must be called before `initialize`.

- `void process()`: Starts processing a collection using a collection reader. The method will block until the CollectionReader finishes processing the entire collection. Throws ResourceProcessException if a CollectionReader has not been provided or initialize has not been called.

- `void collectionProcessingComplete()`: Sends a Collection Processing Complete request to the UIMA-AS Analysis Service. This call is cascaded down to all delegates; however, if a particular delegate is scaled-out, only one of the instances of the delegate will get this call. The method blocks until all of the components that received this call have returned, or a timeout occurs. On success or failure, the application is notified via the statusCallbackListener's collectionProcessComplete() method.

- `void sendAndReceiveCAS(CAS aCAS)`: Send a CAS, wait for response. On success aCAS contains the analysis results. Throws an exception on error. Note that this interface cannot be used to interface to a CAS Multiplier service, because it will block until the parent comes back, and any child CASes will be ignored.

- `String aHandle deploy( String aDeploymentDescriptor, Map anApplicationContext)`: Deploys the UIMA-AS service specified by the given deployment descriptor in this JVM, and returns a handle for this service. The application context map must contain DD2SpringXsltFilePath and SaxonClasspath entries. This call blocks until the service is ready to process requests, or an exception occurs during deployment. If an exception occurs, the callback listener's

- `void undeploy(String aHandle)`: Tells the specified service to terminate. The handle is the same handle that is returned by the corresponding `deploy(...)` method.

- `void stop()`: Stops the asynchronous client. Removes the Cas pool, drops the connection to the UIMA-AS service queue and stops listening on its response queue. Terminates and undeploys any services which have been started with this client.

  This is an asynchronous call, and can be called at any time.

- `void stopProducingCases()`: Send stop signals for all CASes that are currently in process (where the API is expecting responses). If a CAS is a parent of child CASes being produced by a CAS Multiplier, this operation will also signal the CAS Multiplier to stop producing new CASes.

- `void stopProducingCases(String aCasReferenceId)`: send a stop request to a UIMA-AS Service for a particular CAS-id. If that CAS is a parent of child CASes being produced by a CAS Multiplier, this operation will also signal the CAS Multiplier to stop producing new CASes.

4.3. Application Context Map

The application context map is used to pass initialization parameters. These parameters are itemized below.

- DD2SpringXsltFilePath: Required for deploying services.
- SaxonClasspath: Required for deploying services.
- ServerUri: Broker connector for service. Required for initialize.
- Endpoint: Service queue name. Required for initialize.
- Resource Manager: (Optional) a UIMA ResourceManager to use for the client.
- CasPoolSize: Size of Cas pool to create to send to specified service. Default = 1.
- CAS_INITIAL_HEAPSIZE: (Optional) the initial CAS heapsize, in 4-byte words. Default = 500,000.
- Application Name: optional name of the application using this API, for logging.
- Timeout: Process CAS timeout in ms. Default = no timeout.
- GetMetaTimeout: Initialize timeout in ms. Default = 60 seconds.
- CpcTimeout: Collection process complete timeout. Default = no timeout.
- SerializationStrategy:(Optional) xmi or binary serialization. Default = xmi

4.4. Status Callback Listener

Asynchronous events are returned to applications via methods in classes registered to the Client API object with addStatusCallbackListener(). These classes must extend the class org.apache.uima.aae.client.UimaAsBaseCallbackListener.

- `initializationComplete(EntityProcessStatus aStatus)`: The callback used to inform the application that the initialization request has completed. On success aStatus will be null; on failure use the UimaASProcessStatus class to get the details.

- `entityProcessComplete(CAS aCas, EntityProcessStatus aStatus)`: The callback used to inform the application that a processCas request has completed. On success aCAS object will contain result of analysis; on failure the CAS will be in the same state as before it was sent to a service and aStatus will contain the cause of failure. When calling this method, the UIMA AS passes an object of type `UimaASProcessStatus` as a second argument. It extends `EntityProcessStatus` and provides `getCasReferenceId()` method to retrieve a unique id assigned to a CAS. To access this method the user code should implement the following

```
if ( aStatus instanceof UimaASProcessStatus ) {
casReferenceId =
  ((UimaASProcessStatus)aStatus).getCasReferenceId();
}
```

- `collectionProcessComplete(EntityProcessStatus aStatus)`: The callback used to inform the application that the CPC request has completed. On success aStatus will be null; on failure use the `UimaASProcessStatus` class to get the details.

- `onBeforeMessageSend(UimaASProcessStatus status)`: The callback used to inform the application that a CAS is about to be sent to a service. The status object has `getCasReferenceId()` method that returns a unique CAS id assigned by UIMA AS. This reference id may be used to associate arbirary information with a CAS, and is also returned in the callback listener as part of the status object.

4.5. Error Results

Errors are delivered to the callback listeners as an `EntityProcessStatus` or `UimaASProcessStatus` object. These objects provide the methods:

- `isException()`: Indicates the error returned is in the form of exception messages.

- `getExceptions()`: Returns a List of exceptions.

4.6. Asynchronous Client API Usage Scenarios

4.6.1. Instantiating a Client API Object

A client API object must be instantiated for each remote service an application will directly connect with, and a listener class registered in order to process asynchronous events:

```
//create Asynchronous Client API
uimaAsEngine = new BaseUIMAAsynchronousEngine_impl();
uimaAsEngine.addStatusCallbackListener(new MyStatusCallbackListener());
```

4.6.2. Calling an Existing Service

The following code shows how to establish connection to an existing service:

```
//create Map to pass server URI and Endpoint parameters
Map<String,Object> appCtx = new HashMap<String,Object>();
```

```
// Add Broker URI on local machine
appCtx.put(UimaAsynchronousEngine.ServerUri, "tcp://localhost:61616");
// Add Queue Name
appCtx.put(UimaAsynchronousEngine.Endpoint, "RoomNumberAnnotatorQueue");
// Add the Cas Pool Size
appCtx.put(UimaAsynchronousEngine.CasPoolSize, 2);

//initialize
uimaAsEngine.initialize(appCtx);
```

Prepare a Cas and send it to the service:

```
//get an empty CAS from the Cas pool
CAS cas = uimaAsEngine.getCAS();
// Initialize it with input data
cas.setDocumentText("Some text to pass to this service.");
// Send Cas to service for processing
uimaAsEngine.sendCAS(cas);
```

4.6.3. Retrieving Asynchronous Results

Asynchronous events resulting from the process Cas request are passed to the registered listener.

```
// Callback Listener. Receives event notifications from UIMA-AS.
class MyStatusCallbackListener implements UimaASStatusCallbackListener {

// Method called when the processing of a Document is completed.
public void entityProcessComplete(CAS aCas, EntityProcessStatus aStatus) {
  if (aStatus != null && aStatus.isException()) {
    List exceptions = aStatus.getExceptions();
    for (int i = 0; i < exceptions.size(); i++) {
      ((Throwable) exceptions.get(i)).printStackTrace();
      }
    uimaAsEngine.stop();
    return;
    }

// Process the retrieved Cas here
// ...
  }

// Add other required callback methods below...
}
```

4.6.4. Deploying a Service with the Client API

Services can be deployed from a client object independently of any service connection. The main motivation for this feature is to be able to deploy a service, connect to it, and then remove the service when the application is done using it.

```
// create Map to hold required parameters
Map<String,Object> appCtx = new HashMap<String,Object>();
appCtx.put(UimaAsynchronousEngine.DD2SpringXsltFilePath,
           System.getenv("UIMA_HOME") + "/bin/dd2spring.xsl");
```

```
appCtx.put(UimaAsynchronousEngine.SaxonClasspath,
           "file:" + System.getenv("UIMA_HOME") + "/saxon/saxon8.jar");
uimaAsEngine.deploy(service, appCtx);
```

4.7. Undeploying a Service with the Client API

Services can be undeployed from a client object as follows:

```
// create Map to hold required parameters
Map<String,Object> appCtx = new HashMap<String,Object>();
appCtx.put(UimaAsynchronousEngine.DD2SpringXsltFilePath,
           System.getenv("UIMA_HOME") + "/bin/dd2spring.xsl");
appCtx.put(UimaAsynchronousEngine.SaxonClasspath,
           "file:" + System.getenv("UIMA_HOME") + "/saxon/saxon8.jar");
String id = uimaAsEngine.deploy(service, appCtx);
uimaAsEngine.undeploy(id);
```

4.8. Recovering from broker failure

The Client API has a built in recovery strategy to handle cases where a broker fails or becomes unreachable, and then, later becomes available again.

Before sending a new request to a broker, the client checks the state of its connection. If the connection has failed, the client enters a loop where it will attemp to reconnect every 5 seconds. One message is logged to notify this is happening. The recovery attempt stops when the the connection is recovered, or when all UIMA AS clients that are sharing this failed connection, terminate.

During the recovery attempt, any CASes that are submitted via the client APIs will fail or timeout. If the application uses the sendAndReceive() synchronous API, the failure will be delivered by an exception. If the application client uses the sendCAS() asynchronous API, the failure will be delivered via the normal callback listener that the application registered with the UIMA AS client.

4.9. Sample Code

See $UIMA_HOME/examples/src/org/apache/uima/examples/as/RunRemoteAsyncAE.java

Chapter 5. Monitoring, Tuning and Debugging

UIMA AS deployments can involve many separate parts running on many different machines. Monitoring facilities and tools built into UIMA AS help in collecting information on the performance of these parts. You can use the monitoring information to identify deployment issues, such as bottlenecks, and address these with various approaches that alter the deployment choices; this is what we mean by "tuning the deployment".

Monitoring happens in several parts:

- Each node running a JVM hosting UIMA AS services or clients provides JMX information tracking many items of interest.
- UIMA AS services include some of these measurements in the information passed back to its client, along with the returned CAS. This allows clients to collect and aggregate measurements over a cluster of remotely-deployed components.

Tuning a UIMA AS application is done using several approaches:

- changing the topology of the scaleout - for instance, allocating more nodes to some parts, less to others
- adjusting deployment parameters, such as the number of CASes in a CasPool, or the number of threads assigned to do various tasks

In addition, tuning can involve changing the actual analytic algorithms to tune them - but that is beyond the scope of this chapter.

UIMA AS scale out configurations add multithreaded and out-of-order execution complexities to core UIMA applications. Debugging a UIMA AS application is aided by UIMA's modular architecture and an approach that exercises the code gradually from simpler to more complex configurations. Two useful built-in debug features are:

- Java errors at any component level are propagated back to the component originating the request, with a full call chain of UIMA AS components, within colocated aggregate components and across remote services which are shared by multiple clients.
- CASes can be saved before sending to any local or remote delegate and later used to reproduce problems in a simple unit testing environment.

5.1. Monitoring

5.1.1. JMX

JMX (Java Management Extensions) is a standard Java mechanism that is used to monitor and control Java applications. A standard Java tool provided with most Javas, called `jconsole`, is a GUI based application that can connect to a JVM and display the information JMX is providing, and also control the application in application-defined specific ways.

JMX information is provided by a hierarchy of JMX Beans. More background and information on JMX and the jconsole tool is available on the web.

5.1.2. JMX Information from UIMA AS

JMX information is provided by every UIMA AS service or client as it runs. Each item provided is either an instantaneous measurement (e.g. the number of items in a queue) or an accumulating measurement (e.g. the number of CASes processed). Accumulating measures can be reset to 0 using standard JMX mechanisms.

JMX information is provided on a JVM basis; a JVM can be hosting 0 or more UIMA AS Services and/or clients. A UIMA AS Service is defined as a component that connects to a queue and accepts CASes to process. A UIMA AS Client, in contrast, sends CASes to be processed; it can be a top level client, or a UIMA AS Service having one or more AS Aggregate delegates, to which it is sending CASes to be processed.

UIMA AS Services send some of their measurements back to the UIMA AS Clients that sent them CASes; those clients incorporate these measurements into aggregate statistics that they provide. This allows accumulating information among components deployed over many nodes interconnected on a network.

Some JMX measurement items are constant, and document various settings, descriptors, names, etc., in use by the (one or more) UIMA AS services and/or clients running on this JVM.

Some time measurements are associated with running some process. These, where possible, are cpu times, as measured by the thread or threads running the process, using the ThreadMXBean class. On some Javas, thread-based cpu time may not be supported, however. In that case, wall-clock time is used instead.

If the process is multi-threaded, and the cpu has multiple cores, you can get time measurements which exceed the wall clock interval, due to the process consuming cpu time on multiple threads at once.

Timing information not associated with running code, such as idle time, is measured as wall-clock time.

The following sections describe the JMX Beans implemented by UIMA AS. The Notes in the tables include the following flags:

- **inst/acc/const** - instantaneous, accumulating, or constant measurement
- **sent** - sent up to the invoking client with returning CAS

5.1.2.1. UIMA AS Services JMX measures

The next 4 tables detail the JMX measures provided by UIMA AS services.

Service information

Name	Description	Units	Notes
state	The state of the service (Running, Initializing, Disabled, Stopping, Failed)	string	inst
input queueName	The name of the input queue	string	const
reply queueName	The internally generated name of the reply queue	string	const (but could change

Name	Description	Units	Notes
			due to reconnection recovery)
broker URL	The URL of the JMS queue broker	string	const
deployment descriptor	The path to the deployment descriptor for this service	string	const
is CAS Multiplier	is this Service a CAS Multiplier	boolean	const
is top level	is this Service a top level service, meaning that it connects to an input queue on a queue broker	boolean	const
service key	The key name used in the associated Analysis Engine aggregate that specifies this as a delegate	string	const
is Aggregate	is this service an AS Aggregate (i.e., has delegates and is marked async="true")	boolean	const
analysisEngine instance count	The number of replications of the AS Primitive	count	const

Service Performance Measurements

Name	Description	Units	Notes
number of CASes processed	The number of CASes processed by a component	count - CASes	acc
cas deserialization time	The thread time spent deserializing CASes (receiving, either from client, or replies from delegates)	milli seconds	acc
cas serialization time	The thread time spent serializing CASes (sending, either to delegates or back to client)	count - CASes	acc
analysis time	The thread time spent in AS Primitive analytics	milli seconds	acc
idle time	The wall clock time a service has been idle. Measure starts after a reply is sent until the next request is receives, and excludes serialization/deserialization times.	milli seconds	acc
cas pool wait time	The time spent waiting for a CAS to become available in the CAS Pool	milli seconds	acc
shadow cas pool wait time	A shadow cas pool is established for services which are Cas Multipliers. This is the time spent waiting for a CAS to become available in the Shadow CAS Pool.	milli seconds	acc
time spent in CM getNext	The time spent inside Cas Multipliers, getting another CAS. This time (doesn't include / includes ????) the time spent waiting for a CAS to	milli seconds	acc

Name	Description	Units	Notes
	become available in the CAS Pool waiting for a CAS to become available in the CAS Pool		
process thread count	The number of threads available to process requests (number of instances of a primitive)	count	const
reply thread count	The number of threads available to process replies	count	const

Co-located Service Queues

Co-located services use light-weight, internal (not JMS) queues. These have similar measures as are used with JMS queues, and include these measures for both the input queues and the reply (output) queues:

Name	Description	Units	Notes
consumer count	The number of threads configured to read the queue	count	const
dequeue count	The number of CASes that have been read from this queue	count	acc
queue size	The number of CASes in the queue	count	inst

Service Error Measurements

Name	Description	Units	Notes
process Errors	The number of process errors	count	acc
getMetadata Errors	The number of getMetadata errors	count	acc
cpc Errors	The number of Collection Process Complete (cpc) errors	count	acc

5.1.2.2. Application Client information

This section describes monitoring information provided by the UIMA AS Client APIs. Any code that uses the Section 4.1, “Async Client API” [27], such as the example application client `RunRemoteAsyncAE`, will have a set of these JMX measures. Currently no additional tooling (beyond standard tools like `jconsole`) are provided to view these.

Client Measures

Name	Description	Units	Notes
application Name	A user-supplied string identifying the application	string	const
service queue name	The name of the service queue this client connects to	string	const
serialization method	either xmi or binary. This is the serialization the client will use to send CASes to the service, and also tells the service which serialization to use in sending the CASes back.	string	const

Name	Description	Units	Notes
cas pool size	This client's cas pool size, limiting the number of simultaneous outstanding requests in process	count	const
total number of CASes processed	count of the total number of CASes sent from this client. Note: in the case where the service is a Cas Multiplier, the "child" CASes are not included in this count.	count	acc
total time to process	total thread time spent in processing all CASes, including time in remote delegates	milli seconds	acc
average process time	total number of CASes processed / total time to process	milli seconds	inst
max process time	maximum thread time spent in processing a CAS, including time in remote delegates	milli seconds	inst
total serialization time	total thread time spent in serializing, both to delegates (and recursively, to their delegates) and replies back to senders	milli seconds	acc
average serialization time	average thread time spent in serializing a CAS, both to delegates (and recursively, to their delegates) and replies back to senders	milli seconds	inst
max serialization time	maximum thread time spent in serializing a CAS, both to delegates (and recursively, to their delegates) and replies back to senders	milli seconds	inst
total deserialization time	total thread time spent in deserializing, both replies from delegates and CASes from upper level components being sent to lower level ones.	milli seconds	acc
average deserialization time	average thread time spent in deserializing, both replies from delegates and CASes from upper level components being sent to lower level ones.	milli seconds	inst
max deserialization time	maximum thread time spent in deserializing, both replies from delegates and CASes from upper level components being sent to lower level ones.	milli seconds	inst
total idle time	total wall clock time a top-level service thread has been idle since the thread was last used. If there is more than one service thread, this number is the sum.	milli seconds	acc
average idle time	average wall clock time all top-level service threads have been idle since they were last used	milli seconds	inst
max idle time	maximum wall clock time a top-level service thread has been idle since the thread was last used	milli seconds	inst
total time waiting for reply	total wall clock time, measured from the time a CAS is sent to the top-level queue, until that CAS is returned. Any generated CASes from Cas Multipliers are not counted in this measurement.	milli seconds	acc

Name	Description	Units	Notes
average time waiting for reply	average wall clock time from the time a CAS is sent to the reply is received	milli seconds	inst
max time waiting for reply	maximum wall clock time from the time a CAS is sent to the reply is received	milli seconds	inst
total response latency time	total wall clock time, measured from the time a CAS is sent to the top-level queue, including the serialization and deserialization times at the client, until that CAS is returned. Any generated CASes from Cas Multipliers are not counted in this measurement.	milli seconds	acc
average response latency time	average wall clock time, measured from the time a CAS is sent to the top-level queue, including the serialization and deserialization times at the client, until that CAS is returned.	milli seconds	inst
max response latency time	maximum wall clock time, measured from the time a CAS is sent to the top-level queue, including the serialization and deserialization times at the client, until that CAS is returned.	milli seconds	inst
total time waiting for CAS	total wall-clock time spent waiting for a free CAS to be available in the client's CAS pool, before sending the CAS to input queue for the top level service.	milli seconds	acc
average time waiting for CAS	average wall-clock time spent waiting for a free CAS to be available in the client's CAS pool	milli seconds	inst
max time waiting for CAS	maximum wall-clock time spent waiting for a free CAS to be available in the client's CAS pool	milli seconds	inst
total number of CASes requested	total number of CASes fetched from the CAS pool	count	acc

Client Error Measurements

Name	Description	Units	Notes
getMeta Timeout Error Count	number of times a getMeta timed out	count	acc
getMeta Error Count	number of times a getMeta request returned with an error	count	acc
process Timeout Error Count	number of times a process call timed out	count	acc
process Error Count	number of times a process call returned with an error	count	acc

5.2. Logging Sampled JMX information at intervals

A common tuning procedure is to run a deployment for a fairly long time with a typical load, and to see what and where hot spots develop. During this process, it is sometimes useful to convert accumulating measurements into averages, perhaps averages per CAS processed.

UIMA AS includes a monitor component, org.apache.uima.aae.jmx.monitor.JmxMonitor, to sample JMX measures at specified intervals, compute various averages, and write the results into the UIMA Log (or on the console if no log is configured). The monitor program can be automatically enabled for any deployed service by specifying `-D` parameters on the JVM command line which launches the service, or, it can be run stand-alone; when run stand-alone, you provide an argument specifying the JVM it is to connect to to get the JMX information. It only connects to one JVM per run; typically, you would connect it to the top-level service.

The monitor outputs information for that service and its immediate delegates (local or remote); however, it includes information from the complete recursive chain of delegates when computing its measures. You can get detailed monitoring for sub-services by starting or attaching a monitor to those sub-services.

ActiveMQ uses Queue Brokers to manage the JMS queues used by UIMA AS. These brokers have JMX information that is useful in tuning applications. The Monitor program identifies the Queue Broker being used by the service, and connects to it and incorporates information about queue lengths (both the input queue and the reply queue) into its measurements.

5.2.1. Configuring JVM to run the monitor

Specify the following JVM System Variable parameters to configure a UIMA AS Client or Service to enable sampling and logging of JMX measures:

- `-Duima.jmx.monitor.interval=1000` - (default is 1000) specifies the sampling interval in milliseconds
- `-Duima.jmx.monitor.formatter=<CustomFormatterClassName>`
- `-Dcom.sun.management.jmxremote` - enable JMX (only needed for local monitoring, not needed if port is specified)
- `-Dcom.sun.management.jmxremote.port=8009`
- `-Dcom.sun.management.jmxremote.authenticate=false`
- `-Dcom.sun.management.jmxremote.ssl=false`

This configures JMX to run on port 8009 with no authentication, and sets the sampling interval to 1 second, and specifies a custom formatter class name.

There are two `formatter-classes` provided with UIMA AS:

- `org.apache.uima.aae.jmx.monitor.BasicUimaJmxMonitorListener` - this is a multi-line formatter that formats for human-readable output
- `org.apache.uima.aae.jmx.monitor.SingleLineUimaJmxMonitorListener` - this is a formatter that produces one line per interval, suitable for importing into a spreadsheet program.

Both of these log to the UIMA log at the INFO log level.

You can also write your own formatter. The monitor provides an API to plug in a custom formatter for displaying service metrics. A custom formatter must implement JmxMonitorListener interface. See the method `startMonitor` in the class `UIMA_Service` for an example of how custom JMX Listeners are plugged into the monitor.

5.2.2. Running the Monitor program standalone

The monitor program can be started separately and pointed to a running UIMA AS Client or Service. To start the program, invoke Java with the following classpath and parameters:

- ClassPath:
  - %UIMA_HOME%/lib/uimaj-as-activemq.jar
  - %UIMA_HOME%/lib/uimaj-as-core.jar
  - %UIMA_HOME%/lib/uima-core.jar
  - %UIMA_HOME%/apache-activemq-5.4.1/activemq-all-5.4.1.jar
- Parameters:
  - `-Djava.util.logging.config.file=%UIMA_HOME%/config/MonitorLogger.properties` - specifies the logging file where the information is written to
  - `org.apache.uima.aae.jmx.monitor.JmxMonitor` - the class whose main method is invoked
  - `uri` - the URI of the jmx instance to monitor.
  - `interval` - the (optional) sampling interval, in milliseconds (default = 1000)

When run in this manner, it is not (currently) possible to specify the log message formatting class; the multi-line output format is always used.

5.2.3. Monitoring output

The monitoring program combines information from the JMX measures, including the associated Queue Broker, sampling accumulating measurements at the specified sampling interval, and produces the following outputs:

Name	Description	Units
Input queue depth	number of CASes waiting to be processed by a service	count
Reply queue depth	number of CASes returned to the client but not yet picked up by the client	count
CASes processed in interval	Number of CASes processed in this sampling interval	count
Idle time in interval	The total time this service has been idle during this interval	milli seconds

Name	Description	Units
Analysis time in interval	The sum of the times spent in analysis by the service during this interval, including analysis time spent in delegates, recursively	milli seconds
Cas Pool free Cas Count	Number of available CASes in the Cas Pool at the end of the interval	count

In addition to the performance metrics the monitor also provides basic service information:

- Service name
- Is service top level
- Is service remote
- Is service a cas multiplier
- Number of processing threads
- Service uptime (milliseconds)

5.3. Tuning

5.3.1. Tuning procedure

This section is a cookbook of best practices for tuning a UIMA AS deployment. The summary information provided by the Monitor program is used to guide the tuning.

The main metric for detecting an overloaded service is the input queue depth. If it is growing or high, the service is not able to keep up with the load. There are more CASes arriving at the queue than the service can process. Consider increasing number of instances of the services within the JVM (if on a multi-core machine having additional capacity), or deploy additional instances of the service.

The main metric for detecting idle service is the idle time. If it is high, it can indicate that the service is not receiving enough CASes. This can be caused by a bottleneck in the service's client; supporting evidence for this can be a high reply queue depth for the client - indicating the client is overloaded. If the idle time is zero, the service may be saturated; adding more instances could relieve a bottleneck.

A CasPool free Cas Count of 0 can point to a bottleneck in a service's client; supporting evidence for this can be a high idle time. In this case, the service does not have enough CASes in its pool and is forced to wait. Remember that a CAS is not returned to the Service's CAS pool until the client (which can be a parent asynchronous aggregate) signals it can be. A typical reason is a slow client (look for evidence such as a high reply queue depth). Consider incrementing service's Cas pool and check the client's metrics to determine a reason why it is slow.

An asynchronous system must have something that limits the generation of new work to do. CasPools are the mechanism used by UIMA AS to do this. Also, because CASes can have large memory requirements, it is important to limit the number and sizes of CASes in a process.

5.3.2. Tuning Settings

This section has a list of the tuning parameters and a description of what they do and how they interact.

Name	Description
number of services on different machines started	You can adjust the number of machines assigned to a particular service, even dynamically, by just starting / stopping additional servers that specify the same input queue.
number of instances of a service	This is similar to the number of services on different machines started, above, but specifies replication of an AS Primitive within one JVM. This is useful for making use of multi-core machines sharing a common memory - large tables that might be part of the analysis algorithm can be shared by all instances.
CAS pool size	This size limits the number of CASes being processed asynchronously.
casMultiplier poolSize	This size limits the number of CASes generated by a CAS Multiplier that are being processed asynchronously.
Service input queue prefetch	If set greater than 0, allows up to "n" CASes to be pulled into one service provider, at a time. This can increase throughput, but can hurt latency, since one service may have several CASes pulled into it, queued up, while another instance of the service could be "starved" and be sitting there idle.
Specifying async="true"/"false" on an aggregate	The default is false, because there is less overhead (no queues are set up, etc.). Setting this to "true" allows multiple CASes to flow simultaneously in the aggregate.
remoteReplyQueueScaleout	This parameter indicates the number of threads that will be deployed to read from the remote reply queue. Set to > 1 if deserialization time of replies is a bottleneck.

5.4. Debugging

One of the strongest UIMA features is the ability to develop and debug components in isolation from each other, and then to incrementally combine components and scaleout complexity. All that is needed to exercise each configuration are one or more appropriate input CASes.

It is strongly advised to first test UIMA components in the core UIMA environment with a variety of input CASes. If the entire application will not fit in a single process, deploy remote delegates as UIMA AS primitives with only a single instance (see Section 1.4.4.1, “Multiple Instances” [6]), and access them via JMS service descriptors (see Section 1.7, “JMS Service Descriptor” [9]). Run as much input data thru this "single-threaded" configuration as needed to eliminate most "algorithmic" errors and to measure performance against analysis time objectives. Thread safety and analysis ordering issues can then be addressed separately.

Thread safety bugs. Components intended to be run multi-threaded should first be deployed as a multiple instance UIMA AS service (again see Section 1.4.4.1, “Multiple Instances” [6]), and fed their input CASes with a driver capable of keeping all instances busy at the same time. A good application is the sample driver $UIMA_HOME/bin/runRemoteAsyncAE; use the -p argument

to increase the number of outstanding CAS requests sent to the target service. When looking for threading problems try using http://findbugs.sourceforge.net/. In addition to looking for exceptions caused by thread unsafe code, check that the single and multi-threaded analysis results are the same.

Analysis ordering bugs. In a core UIMA aggregate CASes are processed by each delegate in input order. This relationship changes for the same aggregate deployed asynchronously if one of the delegates is replicated, as CASes are progressed in parallel and then progress thru the subsequent aggregate flow in a different order then they are received. Similarly with a delegate CasMultiplier in a core UIMA aggregate each child CAS is processed to completion before the next child CAS is started and the parent CAS is processed last. When running asynchronously the parent CAS can arrive at downstream components ahead of its children because the parent is released from a CasMultiplier immediately after the last child is created. For applications which require all children to be processed before their parent, use the processParentLast flag (see Section 3.9, “Analysis Engine” [20]).

Timing issues. Invariably with complex analytics, some components will be slower and some artifacts will take longer to process than desired. Making performance improvements relies on identifying components running slower than expected and capturing the slow-running artifacts to study in detail.

5.4.1. Error Reporting and Tracing

After the system is scaled out and substantially more data is being processed it is likely that additional errors will occur.

Java errors at any component level are propagated back to the component originating the request (unless suppressed by UIMA AS error handling options, see Section 2.3, “Error handling overview” [12]). The error stack traces the call chain of UIMA AS components, within colocated aggregate components and across remote services which are shared by multiple clients. Some errors can be resolved with this information alone.

If process timeouts are not used (see Section 3.10, “Error Configuration descriptors” [23]) an asynchronous system can hang if one analysis step somewhere in the system has hung. Given many CASes in process at the same time it can be useful to create a custom trace of CAS activity by appropriate logging in **a custom flow controller**. Such logging would have a unique identifier in every CAS, usually a singleton FeatureStructure with a unique String feature. Identifiers for child CASes should include some reference to the CasMultiplier they were created from as well as their parent CAS.

The flow controller is also the ideal place to measure timing statistics for components of interest. Global stats can easily be measured using the time between flow steps, and time thresholds used to flag specific CASes causing problems. Again the unique CAS identifier can be quite useful here.

5.4.2. CAS Logging

Within a UIMA AS asynchronous aggregate, CASes can be saved before sending to any local or remote delegate and later used to reproduce a problem in a simple unit testing environment. Control of CAS logging is done via Java properties:

Property	Description
UIMA_CASLOG_BASE_DIRECTORY	optional; this is the directory under which sub-directories with XmiCas files will be created.

Property	Description
	If not specified, the process's current directory will be the base.
UIMA_CASLOG_COMPONENT_ARRAY	This is a space separated list of delegates keys. If a delegate is nested inside a co-located async aggregate, the name would include the key name of the aggregate, e.g. "someAggName/someDelName". The XmiCas files will then be written into $UIMA_CASLOG_BASE_DIRECTORY/ someAggName-someDelName/
UIMA_CASLOG_TYPE_NAME	optional; this is the name of a FeatureStructure in the CAS containing a unique string to use to name each XmiCas file. If not specified, XmiCas file name will be N.xmi, where N is the time in microseconds since the component was initialized.
UIMA_CASLOG_FEATURE_NAME	optional unless the TYPE_NAME is specified; this parameter gives the string feature to use. If the string value contains one or more "/" characters only the text after the last "/" will be used.
UIMA_CASLOG_VIEW_NAME	optional; if the TYPE_NAME and FEATURE_NAME parameters are specified this string selects the CAS view used to access the FeatureStructure with unique string feature.

Chapter 6. Asynchronous Scaleout Camel Driver

6.1. Overview

Apache Camel[1] is an integration framework based on Enterprise Integration Patterns[2] which uses routes for rule-based message routing and mediation. The camel project has a large number of components which provide access to a wide variety of technologies and are the building blocks of the routes. The Asynchronous Scaleout Camel Driver is a component to integrate UIMA AS into Camel.

6.2. How does it work?

The Asynchronous Scaleout Camel Driver sends the camel message body (without headers) to a specified UIMA AS processing pipeline. Accessing the analysis results which are written into the CAS is not possible from a camel route. There are basically two usage scenarios. The Camel Driver can be used to drive the processing of a UIMA AS cluster in which each server instance runs a cas multiplier to fetch the actual document from a database. In this scenario the camel route only sends an ID of the document to the cas multiplier which does the actual fetching of the document. In the second usage scenario the Camel Driver is used to send a document in a one way fashion to a UIMA AS processing pipeline which then takes care of processing it. In case an error occurs inside the processing pipeline the exception is forwarded to camel and set on the message as response. Error handling is described in the Error handling in Camel[3] documentation.

The camel driver expects a string message body; if it is not of the type string it might be automatically converted by camel type converters. The string message body is set as the CAS's document text. An Analysis Engine calls CAS CAS.getDocumentText() to retrieve the string.

6.3. URI Format

The Asynchronous Scaleout Camel Driver is configured with a configuration string. The configuration string must contain the broker location and name of the JMS queue used to communicate with UIMA AS. It has the following format

```
uimadriver:brokerURL?queue=nameofqueue&CasPoolSize=n&Timeout=t
```

which could for example be specified as

```
uimadriver:tcp://localhost:61616?queue=TextAnalysisQueue
```

. The CasPoolSize parameter is optional but if it is present n must be an integer which is larger than zero, otherwise the UIMA AS default will be used. The Timeout parameter is optional but if present t in milliseconds must zero or larger.

[1] http://camel.apache.org
[2] http://camel.apache.org/enterprise-integration-patterns.html
[3] http://camel.apache.org/error-handling-in-camel.html

6.4. Sample

Camel enables a developer to create quickly all kinds of applications out of existing and custom components. The sample demonstrates how UIMA AS is integrated with other technologies. Readers who are new to camel should read the Getting Started[4] chapter in the camel documentation.

First a simple sample: A user wants to test a UIMA AS processing pipeline, sending it a set of test documents to process. The plain text test documents are located in a folder "/test-data". A camel route for this defined with Java DSL[5] could look like this:

```
from("file://test-data?noop=true").
to("uimadriver:tcp://localhost:61616?queue=TextAnalysisQueue");
```

In the route above the file component sends a message for every file to the uimadriver component. The message contains a reference to the file but not the content of the file itself. The uimadriver component expects a message with string body as input. An internal camel type converter will read in the bytes of the file, decode them into characters with the default platform encoding and then create a string object which is passed to the uimadriver component. The uimadriver then puts the string into a CAS and sends it via the UIMA AS Client API to a processing pipeline. Note that results from the returned CAS cannot be retrieved in a camel route.

A more complex sample. A web site has an area where people can upload pictures. The pictures must be checked for appropriate content. The pictures are pushed to the site via http, stored in a database and assigned to the human controllers to classify them either as appropriate or non-appropriate. That is achieved with an existing camel route and a servlet which receives the images and sends them to the camel route.

```
from("direct:start").
to("imagewriter").
to("jms:queue:HumanPictureAnalysisQueue");
```

The message containing the image is received by the direct:start endpoint, the image is written to a database and replaced with a string identifier by the "imagewriter" component, in the last step the camel jms component posts the identifier on a JMS queue to notify the reception of a new image. The notification is received by a client tool which the human controllers use to classify an image.

To lessen the workload on the human reviewers, a system should automatically classify the pictures and only assign questionable cases to human reviewers. The automatic classification is done by an UIMA Analysis Engine. The AE can mark an image with one of three classes appropriate, non-appropriate and unknown. In the case of unknown the AE is not confident enough which of the first two classes is correct. To be scalable, the processing pipeline is hosted by UIMA AS and contains three AEs, one to fetch the image from the database, a classification AE and an AE to write the class of the image back to the database. The first AE is typical a cas multiplier and receives a CAS which only contains the string identifier but not the actual image. The cas muliplier uses the identifer to fetch the image from the database and outputs a new CAS with the actual image. The Camel route blocks until the CAS is processed by the following two AEs and depending on the class in the database the picture is assigned to a human controller or not.

[4] http://camel.apache.org/book-getting-started.html
[5] http://camel.apache.org/dsl.html

```
from("direct:start").
to("imagewriter").
to("uimadriver:tcp://localhost:61616?queue=UimaPictureAnalysisQueue").
to("class-retriever").
// filters messages with class appropriate and non-appropriate
filter(header("picture-class").isEqualTo("unkown")).
to("jms:queue:HumanPictureAnalysisQueue");
```

The first part is identical, after the imagewriter the string identifier is send to the UIMA AS processing pipeline which writes the image class back to the database. The class is retrieved with the custom class-retriever component and written to a message header field, only if the class is unknown the image is assigned for human classification.

Note: instead of using a CAS multiplier, a more straight-forward approach would use just one CAS, having 2 views: one view would contain the string identifier of the image, and the other view would have the image to be analyzed (or a reference to it in the DB).

6.5. Implementation

The Asynchronous Scaleout Camel Driver is a typical camel component. The camel documentation Writing Components[6] describes how camel components are written. The source code can be found in the uimaj-as-camel project. The implementation defines an asynchronous producer endpoint, which is implemented in the `org.apache.uima.camel.UimaAsProducer` class. The `UimaAsProducer.process` method gets the string body of the message, wraps it in a CAS object and sends it to UIMA AS. Since the producer is asynchronous the camel message is registered with the reference id of the sent CAS in an intermediate map, when the CAS comes back from UIMA AS, the camel message is looked up with the reference id of the CAS and the processing of the camel message is completed. For further details please read the `UimaAsProducer` implementation code.

[6] http://camel.apache.org/writing-components.html