Distributed UIMA Cluster Computing

Written and maintained by the Apache
UIMA^{TM}Development Community

Version 1.0.0

Publication date: 14-01-2014

Table of Contents

List of Figures

Part I

DUCC Concepts

Chapter 1

DUCC Overview

1.1 What is DUCC?

DUCC stands for Distributed UIMA Cluster Computing. DUCC is a cluster management system providing tooling, management, and scheduling facilities to automate the scale-out of applications written to the UIMA framework.

Core UIMA provides a generalized framework for applications that process unstructured information such as human language, but does not provide a scale-out mechanism. UIMA-AS provides a scale-out mechanism to distribute UIMA pipelines over a cluster of computing resources, but does not provide job or cluster management of the resources. DUCC defines a formal job model that closely maps to a standard UIMA pipeline. Around this job model DUCC provides cluster management services to automate the scale-out of UIMA pipelines over computing clusters.

1.2 DUCC Job Model

The Job Model defines the steps necessary to scale-up a UIMA pipeline using DUCC. The goal of DUCC is to allow the application logic to be unchanged.

The DUCC Job model consists of standard UIMA components: a Collection Reader (CR), a CAS Multiplier (CM), application logic as implemented one or more Analysis Engines (AE), and a CAS Consumer (CC). In theory, any CR, or CM will work with DUCC, but DUCC is all about scale-out. In order to achieve good scale-out these components must be constructed in a specific way.

The Collection Reader builds input CASs and forwards them to the UIMA pipelines. In the DUCC model, the CR is run in a process separate from the rest of the pipeline. In fact, in all but the smallest clusters it is run on a different physical machine than the rest of the pipeline. To achieve scalability, the CR must create very small CASs that do not contain application data, but which contain references to data; for instance, file names. Ideally, the CR should be runnable in a process not much larger than the smallest Java virtual machine. Later sections demonstrate methods for achieving this.

Each pipeline must contain at least one CAS Multiplier which receives the CASs from the CR. The CMs encapsulate the knowledge of how to receive the data references in the small CASs received from the CRs and deliver the referenced data to the application pipeline. DUCC packages the CM, AE(s), and CC into a single process, multiple instances of which are then deployed over the cluster.

DUCC does not provide any mechanism for receiving output CASs. Each application must supply its own CAS Consumer which serializes the output of the pipelines for consumption by other entities (as serialized CASs, for example).

A DUCC job therefore consists of a small specification containing the following items:

- The name of a resource containing the CR descriptor.

- The name of a resource containing the CM descriptor.
- The name of a resource containing the AE descriptor.
- The name of a resource containing the CC descriptor.
- Other information required to parametrize the above and identify the job such as log directory, working directory, desired scale-out, etc. These are described in detail in subsequent sections.

On job submission, DUCC examines the job specification and automatically creates a scaled-out UIMA-AS service with a single process executing the CR as a UIMA-AS client and and as many processes as possible executing the combined CM, AE, and CC pipeline as UIMA-AS service instances.

DUCC provides other facilities in support of scale-out:

- The ability to reserve all or part of a node in the cluster.
- Automated management of services required in support of jobs.
- The ability to schedule and execute arbitrary processes on nodes in the cluster.
- Debugging tools and support.
- A web server to display and manage work and cluster status.
- A CLI and a Java API to support the above.

1.3 DUCC From UIMA to Full Scale-out

In this section we demonstrate the progression of a simple UIMA pipeline to a fully scaled-out job running under DUCC.

UIMA Pipelines A normal UIMA pipeline contains a Collection Reader, one or more Analysis Engines connected in a pipeline, and a CAS Consumer as shown in Figure 1.1.

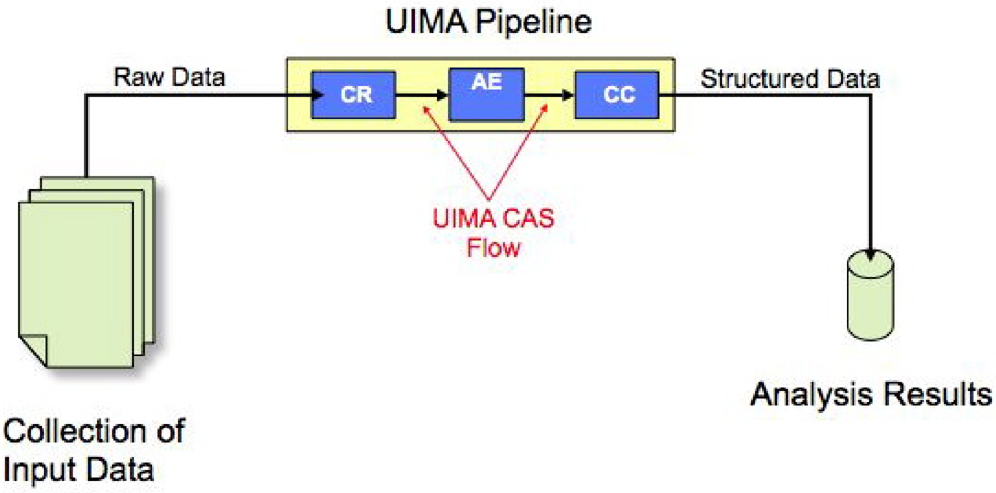


Figure 1.1: Standard UIMA Pipeline

UIMA-AS Scaled Pipeline With UIMA-AS the CR is separated into a discrete process and a CAS Multiplier is introduced into the pipeline as an interface between the CR and the pipeline, as shown in Figure 1.2 below. Multiple pipelines are serviced by the CR and are scaled-out over a computing cluster. The difficulty with this model is that each user is individually responsible for finding and scheduling computing nodes, installing communication software such as ActiveMQ, and generally managing the distributed job and associated hardware.

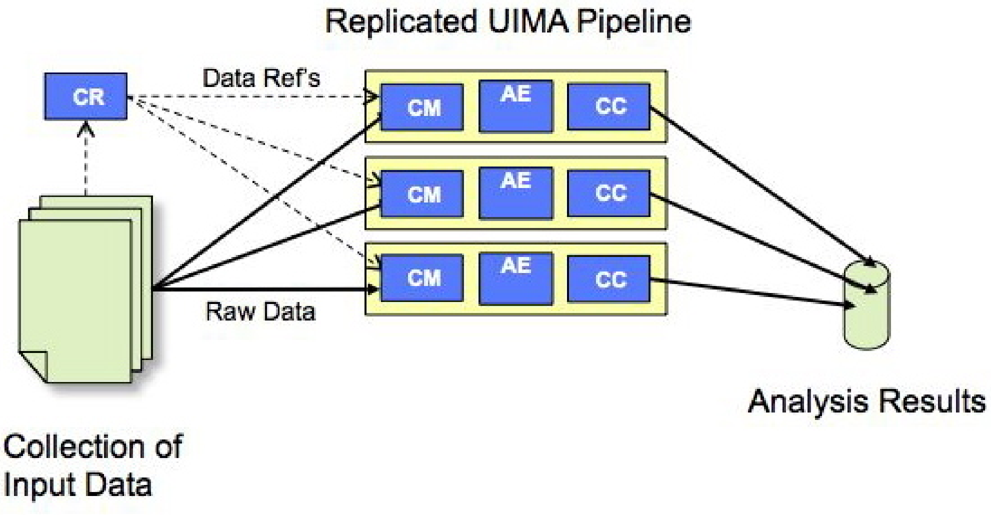


Figure 1.2: UIMA Pipeline As Scaled by UIMA-AS

UIMA-AS Pipeline Scaled By DUCC DUCC is a UIMA and UIMA-AS-aware cluster manager. To scale out work under DUCC the developer tells DUCC what the parts of the application are, and DUCC does the work to build the scale-out via UIMA/AS, to find and schedule resources, to deploy the parts of the application over the cluster, and to manage the jobs while it executes.

On job submission, the DUCC Command Line Interface (CLI) inspects the XML defining the analytic and generates a UIMA-AS Deployment Descriptor (DD). The DD establishes some number of pipeline threads per process (as indicated in the DUCC job parameters), and generates job-unique queues.

Under DUCC, the Collection Reader is executed in a process called the Job Driver (or JD). The pipelines are executed in one or more processes called Job Processes (or JPs). The JD process provides a thin wrapper over the CR to enable communication with DUCC. The JD uses the CR to implement a UIMA-AS client delivering CASs to the multiple (scaled-out) pipelines, shown in Figure 1.3 below.

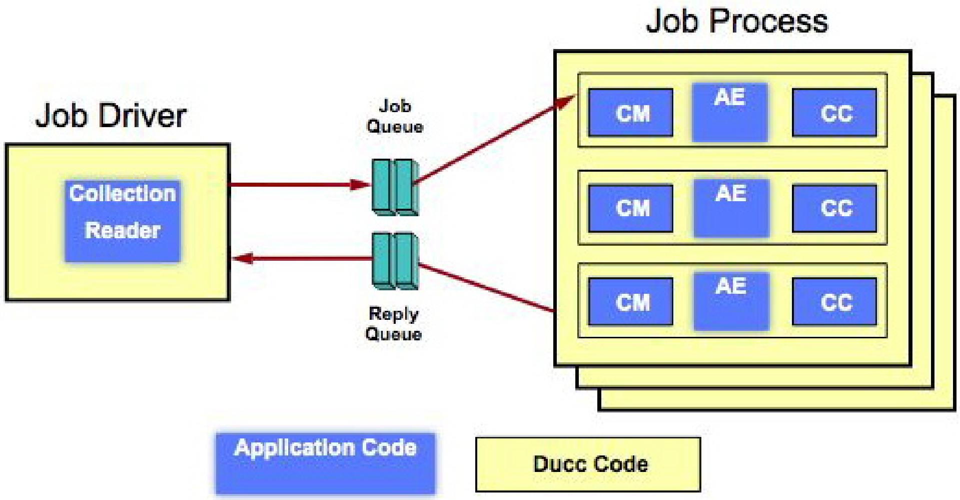


Figure 1.3: UIMA Pipeline As Automatically Scaled Out By DUCC

UIMA-AS Pipeline with User-Supplied DD Scaled By DUCC Application programmers may supply their own Deployment Descriptors to control intra-process threading and scale-out. If a DD is supplied in the job parameters, DUCC will use this instead of generating one as depicted in Figure 1.4 below.

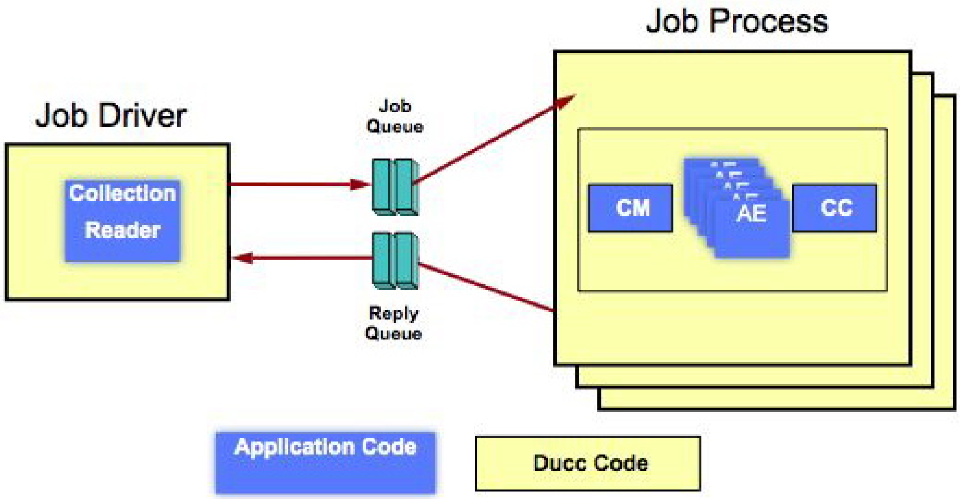


Figure 1.4: UIMA Pipeline With User-Supplied DD as Automatically Scaled Out By DUCC

1.4 Error Management

DUCC provides a number of facilities to assist error management:

- DUCC uses the UIMA-AS error-handling facilities to reflect errors from the Job Processes to the Job Drivers. The JD wrappers implement logic to enforce error thresholds, to identify and log errors, and to reflect job problems in the DUCC Web Server. All error thresholds are configurable both globally and on a per-job basis.

- Error and timeout thresholds are implemented for both the initialization phase of a pipeline and the execution phase.
- Retry-after-error is supported: if a process has a failure on some CAS after initialization is successful, the process is terminated and all affected CASs are retried, up to some configurable threshold.
- DUCC ensures that processes can successfully initialize before fully scaling out a job, to ensure a cluster is not overwhelmed with errant processes.
- Various error conditions encountered while a job is running will prevent the errant job from continuing scale out, and can result in termination of the job.

1.5 Cluster and Job Management

DUCC supports management of multiple jobs and multiple users in a distributed cluster:

Multiple User Support DUCC runs all work under the identity of the submitting user. Logs are written with the user's credentials into the user's file space designated at job submission.

Fair-Share Scheduling DUCC provides a Fair-Share scheduler to equitably share resources among multiple users. The scheduler also supports semi-permanent reservation of full or partial machines.

Service Management DUCC provides a Service Manager capable of automatically starting, stopping, and otherwise managing and querying both UIMA-AS and non-UIMA-AS services in support of jobs.

Job Lifetime Management and Orchestration DUCC includes an Orchestrator to manage the lifetimes of all entities in the system.

Node Sharing DUCC allocates processes from one or more users on a node, each with a specified amount of memory. DUCC's preferred mechanism for constraining memory use is Linux Control Groups, or CGroups. For nodes that do not suport CGroups, DUCC agents monitor RAM use and kill processes that exceed their share size by a settable fudge factor.

DUCC Agents DUCC Agents manage each node's local resources and all processes started by DUCC. Each node in a cluster has exactly one Agent. The Agent

- Monitors and reports node capabilities (memory, etc) and performance data (CPU busy, swap, etc).
- Starts, stops, and monitors all processes on behalf of users.
- Patrols the node for "foreign" (non-DUCC) processes, reporting them to the Web Server, and optionally reaping them.
- Ensures job processes do not exceed their declared memory requirements through the use of Linux Cgroups.

DUCC Web server DUCC provides a web server displaying all aspects of the system:

- All jobs in the system, their current state, resource usage, etc.
- All reserved resources and associated information (owner, etc.), including the ability to request and cancel reservations.
- All services, including the ability to start, stop, and modify service definitions.
- All nodes in the system and their status, usage, etc.
- The status of all DUCC management processes.
- Access to documentation.

Cluster Management Support DUCC provides system management support to:

- Start, stop, and query full DUCC systems.
- Start, stop, and quiesce individual DUCC components.

- Add and delete nodes from the DUCC system.
- Discover DUCC processes (e.g. after partial failures).
- Find and kill errant job processes belonging to individual users.
- Monitor and display inter-DUCC messages.

1.6 Security Measures

The following DUCC security measures are provided:

command line interface The CLI employs HTTP to send requests to the DUCC controller. The CLI creates and employs public and private security keys in the user's home directory for authentication of HTTP requests. The controller validates requests via these same security keys.

webserver The webserver facilitates operational control and therefore authentication is desirable.

user Each user has the ability to control certain aspects of only his/her active submissions.

admin Each administrator has the ability to control certain aspects of any user's active submissions, as well as modification of some DUCC operational characteristics.

A simple interface is provided so that an installation can plug-in a site specific authentication mechanism comprising userid and password.

ActiveMQ TBD.

1.7 Security Issues

The following DUCC security issues should be considered:

submit transmission 'sniffed' In the event that the DUCC submit command is 'sniffed' then the user authentication mechanism is compromised and user masquerading is possible. That is, the userid encryption mechanism can be exploited such that user A can submit a job pretending to be user B.

user *ducc* password compromised In the event that the *ducc* user password is compromised then the root privileged command **ducc_ling** can be used to become any other user except root.

user *root* password compromised In the event that the *root* user password is compromised DUCC provides no protection. That is, compromising the root user is equivalent to compromising the DUCC user password.

Chapter 2

Glossary

Autostarted Service An autostarted service is a registered service that is started automatically by DUCC when the DUCC system is booted.

Dependent service or job A dependent service or job is a service or job that specifies one or more service dependencies in their job specification. The service or job is dependent upon the referenced service being operational before being started by DUCC.

DUCC Distributed UIMA Cluster Computing.

Registered service A registered service is a service that is registered with DUCC. DUCC saves the service specification and fully manages the service, insuring it is running when needed, and shutdown when not.

Service Instance A service instance is one physical process which runs a CUSTOM or UIMA-AS service. UIMA-AS services are usually scaled-out with multiple instances implementing the same underlying service logic.

Orchestrator (OR) The Orchestrator manages the life cycle of all entities within DUCC.

Process Manager (PM) The Process Manager coordinates distribution of work among the Agents.

Resource Manager (RM) The Resource Manager schedules physical resources for DUCC work.

Service Endpoint In DUCC, the service endpoint provides a unique identifier for a service. In the case of UIMA-AS services, the endpoint also serves as a well-known address for contacting the service.

Service Manager (SM) The Service Manager manages the life-cycles of UIMA-AS and CUSTOM services. It coordinates registration of services, starting and stopping of services, and ensures that services are available and remain available for the lifetime of the jobs.

Agent DUCC Agent processes run on every node in the system. The Agent receives orders to start and stop processes on each node. Agents monitors nodes, sending heartbeat packets with node statistics to interested components (such as the RM and web-server). If CGroups are installed in the cluster, the Agent is responsible for managing the CGroups for each job process. All processes other than the DUCC management processes are are managed as children of the agents.

DUCC-MON DUCC-MON is the DUCC web-server.

Job Driver (JD) The Job Driver is a thin wrapper that encapsulates a Job's Collection Reader. The JD executes as a process that is scheduled and deployed by DUCC.

Job Process (JP) The Job Process is a thin wrapper that encapsulates a job's pipeline components. The JP executes in a process that is scheduled and deployed by DUCC.

Job specification The Job Specification is a collection of properties that describe work to be scheduled and deployed by DUCC. It identifies the UIMA components (CR, AE, etc) that comprise the job and the system-wide properties of the job (CLASSPATHs, RAM requirements, etc).

Job A DUCC job consists of the components required to deploy and execute a UIMA pipeline over a computing cluster. It consists of a JD to run the Collection Reader, a set of JPs to run the UIMA AEs, and a Job Specification to describe how the parts fit together.

Share Quantum The DUCC scheduler abstracts the nodes in the cluster as a single large conglomerate of resources: memory, processor cores, etc. The scheduler logically decomposes the collection of resources into some number of equal-sized atomic units. Each unit of work requiring resources is apportioned one or more of these atomic units. The smallest possible atomic unit is called the *share quantum*, or simply, *share*.

Process A process is one physical process executing on a machine in the DUCC cluster. DUCC jobs are comprised of one or more processes (JDs and JPs). Each process is assigned one or more *shares* by the DUCC scheduler.

Weighted Fair Share A weighted fair share calculation is used to apportion resources equitably to the outstanding work in the system. In a non-weighted fair-share system, all work requests are given equal consideration to all resources. To provide some ("more important") work more than equal resources, weights are used to bias the allotment of shares in favor of some classes of work.

Work Items A DUCC work item is one unit of work to be completed in a single DUCC process. It is usually initiated by the submission of a single CAS from the JD to one of the JPs. It could be thought of as a single "question" to be answered by a UIMA analytic, or a single "task" to complete. Usually each DUCC JP executes many work items per job.

$DUCC_HOME The root of the installed DUCC runtime, e.g. /home/ducc/ducc_runtime. It need not be set in the environment, although the examples in this document assume that it has been.

Part II

Ducc Users Guide

Chapter 3

Command Line Interface

Overview The DUCC CLI is the primary means of communication with DUCC. Work is submitted, work is canceled, work is monitored, and work is queried with this interface.

All parameters may be passed to all the CLI commands in the form of Unix-like "long-form" (key, value) pairs, in which the key is proceeded by the characters "−−". As well, the parameters may be saved in a standard Java Properties file, without the leading "−−" characters. Both a properties file and command-line parameters may be passed to each CLI. When both are present, the parameters on the command line take precedence. Take, for example the following simple job properties file, call it `1.job`, where the environment variable "DH" has been set to the location of $DUCC_HOME.

```
description                      Test job 1

classpath                        ${DH}/lib/uima-ducc/examples/*
environment                      AE_INIT_TIME=5 AE_INIT_RANGE=5 LD_LIBRARY_PATH=/a/nother/path
scheduling_class                 normal

driver_descriptor_CR             org.apache.uima.ducc.test.randomsleep.FixedSleepCR
driver_descriptor_CR_overrides jobfile=${DH}/lib/examples/simple/1.inputs compression=10
error_rate=0.0

driver_jvm_args                  -Xmx500M

process_descriptor_AE            org.apache.uima.ducc.test.randomsleep.FixedSleepAE
process_memory_size              4
process_jvm_args                 -Xmx100M
process_thread_count             2
process_per_item_time_max        5
process_deployments_max          999
```

This can be submitted, overriding the scheduling class and memory, thus:

```
ducc_job_submit --specification 1.job --process_memory_size 16 --scheduling_class high
```

The DUCC CLI parameters are now described in detail.

3.1 The DUCC Job Descriptor

The DUCC Job Descriptor includes properties to enable automated management and scale-out over large computing clusters. The job descriptor includes

- References to the various UIMA components required by the job (CR, CM, AE, CC, and maybe DD)
- Scale-out requirements: number of processes, number of threads per process, etc
- Environment requirements: log directory, working directory, environment variables, etc,
- JVM parameters
- Scheduling class
- Error-handling preferences: acceptable failure counts, timeouts, etc
- Debugging and monitoring requirements and preferences

3.2 Operating System Limit Support

The CLI supports specification of operating system limits applied to the various job processes. To specify a limit, pass the name of the limit and its value in the *environment* specified in the job. Limits are named with the string "DUCC_RLIMIT_name" where "name" is the name of a specific limit. Supported limits include:

- DUCC_RLIMIT_CORE
- DUCC_RLIMIT_CPU
- DUCC_RLIMIT_DATA
- DUCC_RLIMIT_FSIZE
- DUCC_RLIMIT_MEMLOCK
- DUCC_RLIMIT_NOFILE
- DUCC_RLIMIT_NPROC
- DUCC_RLIMIT_RSS
- DUCC_RLIMIT_STACK
- DUCC_RLIMIT_AS
- DUCC_RLIMIT_LOCKS
- DUCC_RLIMIT_SIGPENDING
- DUCC_RLIMIT_MSGQUEUE
- DUCC_RLIMIT_NICE
- DUCC_RLIMIT_STACK
- DUCC_RLIMIT_RTPRIO

See the Linux documentation for details on the meanings of these limits and their values.

For example, to set the maximum number of open files allowed in any job process, specify an environment similar to this when submitting the job:

```
ducc_job_submit .... --environment="DUCC_RLIMT_NOFILE=1024" ...
```

3.3 Command Line Forms

The Command Line Interface is provided in several forms:

1. A wrapper script around the uima-ducc-cli.jar.
2. Direct invocation of each command's `class` with the `java` command.

When using the scripts the full execution environment is established silently. When invoking a command's `class` directly, the java `CLASSPATH` must include the uima-ducc-cli.jar, as illustrated in the wrapper scripts.

3.4 DUCC Commands

The following commands are provided:

ducc_submit Submit a job for execution.

ducc_cancel Cancel a job in progress.

ducc_reserve Request a reservation of full or partial machines.

ducc_unreserve Cancel a reservation.

ducc_monitor Monitor the progress of a job that is already submitted.

ducc_process_submit Submit an arbitrary process (managed reservation) for execution.

ducc_process_cancel Cancel an arbitrary process.

ducc_services Register, unregister, start, stop, modify, and query a service.

ducc_view_perf Fetch performance data from the log and history files for analysis by spreadsheets, etc.

viaducc This is a script wrapper to facilitate execution of Eclipse workspaces as DUCC jobs as well as general execution of arbitrary processes in DUCC-managed resources.

The next section describes these commands in detail.

3.5 ducc_submit

The source for this section is ducc_duccbook/documents/part-user/cli/submit.xml.

Description: The submit CLI is used to submit work for execution by DUCC. DUCC assigns a unique id to the job and schedules it for execution. The submitter may optionally request that the progress of the job is monitored, in which case the state of the job as it progresses through its lifetime is printed on the console.

Usage:

Script wrapper $DUCC_HOME/bin/ducc_submit *options*

Java Main java -cp $DUCC_HOME/lib/uima-ducc-cli.jar org.apache.uima.ducc.cli.DuccJobSubmit *options*

Options:

−−**all_in_one** <**local** | **remote** > Run driver and pipeline in single process. If *local* is specified, the process is executed on the local machine, for example, in the current Eclipse session. If *remote* is specified, the jobs is submitted to DUCC as a *manged reservation* and run on some (presumably larger) machine allocated by DUCC.

−−**attach_console** If specified, redirect remote stdout and stderr to the local submitting console.

−−**cancel_on_interrupt** . If the job is started with −−wait_for_completion, this option causes the job to be canceled if the submit command is terminated, e.g., with CTL-C. If −−cancel_job_on_interrupt is not specified, the job monitor will be terminated but the job will continue to run.

If −−wait_for_completion is not specified this option is ignored.

−−**classpath** The CLASSPATH used for the job. If specified, this is used for both the Job Driver and each Job Process. If not specified the CLASSPATH found by the underlying `DuccJobSubmit.main()` method is used.

−−**classpath_order [user-before-ducc | ducc-before-user]** When DUCC deploys a process, set the user-supplied CLASSPATH before DUCC-supplied CLASSPATH, or the reverse. The default is ducc-before-user.

−−**debug** Enable debugging messages. This is primarily for debugging DUCC itself.

−−**description [text]** The text is any string used to describe the job. It is displayed in the Web Server. When specified on a command-line the text usually must be surrounded by quotes to protect it from the shell. The default is "none".

−−**driver_debug [debug-port]** Append JVM debug flags to the JVM arguments to start the JobDriver in remote debug mode. The remote process debugger will attempt to contact the specified port.

−−**driver_descriptor_CR [descriptor.xml]** This is the XML descriptor for the Collection Reader. This descriptor is a resource that is searched for in the filesystem or Java classpath as described in the notes below.

−−**driver_descriptor_CR_overrides [list]** This is the Job Driver collection reader configuration overrides. They are specified as name/value pairs in a comma-delimited list. For example:

```
--driver_descriptor_CR_overrides name1=value1,name2=value2...
```

−−**driver_exception_handler [classname]** This specifies a developer-supplied exception handler for the Job Driver. It must implement org.apache.uima.ducc.common.jd.plugin.IJdProcessExceptionHandler or extend org.apache.uima.ducc.common.jd.plugin.AbstractJdProcessExceptionHandler. A default handler is provided.

−−**driver_jvm_args [list]** This specifies extra JVM arguments to be provided to the Job Driver process. It is a blank-delimited list of strings. Example:

```
--driver_jvm_args -Xmx100M -Xms50M
```

Note: When used as a CLI option, the list must usually be quoted to protect it from the shell.

−−**environment [env vars]** Blank-delimited list of environment variable assignments. If specified, this is used for all DUCC processes in the job. Example:

```
--environment TERM=xterm DISPLAY=:1.0
```

Additional entries may be copied from the user's environment based on the setting of

```
ducc.submit.environment.propagated
```

in the global DUCC configuration ducc.properties.

Note: When used as a CLI option, the environment string must usually be quoted to protect it from the shell.

−−**help** Prints the usage text to the console.

−−**jvm [path-to-java]** States the JVM to use. If not specified, the same JVM used by the Agents is used. This is the full path to the JVM, not the JAVA_HOME. Example:

```
--jvm /share/jdk1.6/bin/java
```

−−**log_directory [path-to-log-directory]** This specifies the path to the directory for the user logs. If not specified, the default is $HOME/ducc/logs. Example:

```
--log_directory /home/bob
```

Within this directory DUCC creates a sub-directory for each job, using the unique numerical ID of the job. The format of the generated log file names as described here.

Note: Note that −−log_directory specifies only the path to a directory where logs are to be stored. In order to manage multiple processes running in multiple machines, sub-directory and file names are generated by DUCC and may not be directly specified.

−−**process_DD [DD descriptor]** This specifies a UIMA Deployment Descriptor for the job processes for DD-style jobs. This is mutually exclusive with −−process_descriptor_AE, −−process_descriptor_CM, and −−process_descriptor_CC. This descriptor is a resource that is searched for in the filesystem or Java classpath as described in the notes below. For example:

```
--process_DD /home/billy/resource/DD_foo.xml
```

−−**process_debug [debug-port]** Append JVM debug flags to the JVM arguments to start the Job Process in remote debug mode. The remote process will start its debugger and attempt to contact the debugger (usually Eclipse) on the specified port.

−−**process_deployments_max [integer]** This specifies the maximum number of Job Processes to deploy at any given time. If not specified, DUCC will attempt to provide the largest number of processes within the constraints of fair_share scheduling and the amount of work remaining. in the job. Example:

```
--process_deployments_max 66
```

−−**process_descriptor_AE [descriptor]** This specifies the Analysis Engine descriptor to be deployed in the Job Processes. This descriptor is a resource that is searched for in the filesystem or Java classpath as described in the notes below. It is mutually exclusive with −−process_DD For example:

```
--process_descriptor_AE /home/billy/resource/AE_foo.xml
```

−−**process_descriptor_AE_overrides [list]** This specifies AE overrides. It is a comma-delimited list of name/value pairs. Example:

```
--process_descriptor_AE_Overrides name1=value1,name2=value2
```

−−**process_descriptor_CC [descriptor]** This specifies the CAS Consumer descriptor to be deployed in the Job Processes. This descriptor is a resource that is searched for in the filesystem or Java classpath as described in the notes below. It is mutually exclusive with −−process_DD For example:

```
--process_descriptor_CC /home/billy/resourceCCE_foo.xml
```

−−**process_descriptor_CC_overrides [list]** This specifies CC overrides. It is a comma-delimited list of name/value pairs. Example:

```
--process_descriptor_CC_overrides name1=value1,name2=value2
```

−−**process_descriptor_CM [descriptor]** This specifies the CAS Multiplier descriptor to be deployed in the Job Processes. This descriptor is a resource that is searched for in the filesystem or Java classpath as described in the notes below. It is mutually exclusive with −−process_DD For example:

```
--process_descriptor_CM /home/billy/resource/CM_foo.xml
```

−−**process_descriptor_CM_overrides [list]** This specifies CM overrides. It is a comma-delimited list of name/value pairs. Example:

```
--process_descriptor_CM_overrides name1=value1,name2=value2
```

−−**process_failures_limit [integer]** This specifies the maximum number of individual Job Process (JP) failures allowed before killing the job. The default is twenty(20). If this limit is exceeded over the lifetime of a job DUCC terminates the entire job.

```
--process_failures_limit 23
```

−−process_initialization_failures_cap [integer] This specifies the maximum number of failures during a UIMA process's initialization phase. If the number is exceeded the system will allow processes which are already running to continue, but will assign no new processes to the job. The default is ninety-nine(99). Example:

```
--process_initialization_failures_cap 62
```

Note that the job is NOT killed if there are processes that have passed initialization and are running. If this limit is reached, the only action is to not start new processes for the job.

−−process_initialization_time_max [integer] This is the maximum time a process is allowed to remain in the "initializing" state, before DUCC terminates it. The error counts as an initialization error towards the initialization failure cap.

−−process_jvm_args [list] This specifies additional arguments to be passed to all of the job processes as a blank-delimited list of strings. Example:

```
--process_jvm_args -Xmx400M -Xms100M
```

Note: When used as a CLI option, the arguments must usually be quoted to protect them from the shell.

−−process_memory_size [size] This specifies the maximum amount of RAM in GB to be allocated to each Job Process. This value is used by the Resource Manager to allocate resources.

−−process_per_item_time_max [integer] This specifies the maximum time in minutes that the Job Driver will wait for a Job Processes to process a CAS. If a timeout occurs the process is terminated and the CAS marked in error (not retried). If not specified, the default is 1 minute. Example:

```
--process_per_item_time_max 60
```

−−process_thread_count [integer] This specifies the number of threads per process to be deployed. It is used by the Resource Manager to determine how many processes are needed, by the Job Process wrapper to determine how many threads to spawn, and by the Job Driver to determine how many CASs to dispatch. If not specified, the default is 4. Example:

```
--process_thread_count 7
```

−−scheduling_class [classname] This specifies the name of the scheduling class the RM will use to determine the resource allocation for each process. The names of the classes are installation dependent. If not specified, the default is taken from the global DUCC configuration ducc.properties. Example:

```
--scheduling_class normal
```

−−service_dependency[list] This specifies a blank-delimited list of services the job processes are dependent upon. Service dependencies are discussed in detail here. Example:

```
--service_dependency UIMA-AS:Service1:tcp:host1:61616 UIMA-AS:Service2:tcp:host2:123
```

−−specification, −f [file] All the parameters used to submit a job may be placed in a standard Java properties file. This file may then be used to submit the job (rather than providing all the parameters directory to submit). The leading −− is omitted from the keywords.

For example,

```
ducc_submit --specification job.props
ducc_submit -f job.props
```

where job.props contains:

```
working_directory                    = /home/bob/projects/ducc/ducc_test/test/bin
process_failures_limit               = 20
driver_descriptor_CR                 = org.apache.uima.ducc.test.randomsleep.FixedSleepCR
environment                          = AE_INIT_TIME=10000 LD_LIBRARY_PATH=/a/bogus/path
log_directory                        = /home/bob/ducc/logs/
process_thread_count                 = 1
driver_descriptor_CR_overrides       = jobfile:../simple/jobs/1.job,compression:10
process_initialization_failures_cap = 99
process_per_item_time_max            = 60
driver_jvm_args                      = -Xmx500M
process_descriptor_AE                = org.apache.uima.ducc.test.randomsleep.FixedSleepAE
classpath                            = /home/bob/duccapps/ducky_process.jar
description                          = ../simple/jobs/1.job[AE]
process_jvm_args                     = -Xmx100M -DdefaultBrokerURL=tcp://localhost:61616
scheduling_class                     = normal
process_memory_size                  = 15
```

Note that properties in a specifications file may be overridden by other command-line parameters, as discussed here.

−−**suppress_console_log** If specified, suppress creation of the log files that normally hold the redirected stdout and stderr.

−−**time-stamp** If specified, messages from the submit process are timestamped. This is intended primarily for use with a monitor with –wait_for_completion.

−−**wait_for_completion** If specified, the submit command monitors the job and prints periodic state and progress information to the console. When the job completes, the monitor is terminated and the submit command returns.

−−**working_directory** This specifies the working directory to be set by the Job Driver and Job Process processes. If not specified, the current directory is used.

Notes: When searching for UIMA XML resource files such as descriptors, DUCC searches either the filesystem or Java classpath according to the following rules:

1. If the resource ends in .xml it is assumed the resource is a file in the filesystem and the path is either an absolute path or a path relative to the specified working directory.
2. If the resource does not end in .xml, it is assumed the resource is in the Java classpath. DUCC creates a resource name by replacing the ”.” separators with ”/” and appending ”.xml”.

3.6 ducc_cancel

Description: The cancel CLI is used to cancel a job that has previously been submitted but which has not yet completed.

Usage:

Script wrapper $DUCC_HOME/bin/ducc_cancel *options*

Java Main java -cp $DUCC_HOME/lib/uima-ducc-cli.jar org.apache.uima.ducc.cli.DuccJobCancel *options*

Options:

−−**debug** Prints internal debugging information, intended for DUCC developers or extended problem determination.

−−**id** [**jobid**] The ID is the id of the job to cancel.

−−**reason** [**quoted string**] Optional. This specifies the reason the job is canceled for display in the web server. Note that the shell requires a quoted string. Example:

```
ducc_cancel --id 12 --reason "This is a pretty good reason."
```

−−**dpid** [**pid**] If specified only this DUCC process will be canceled. If not specified, then entire job will be canceled. The *pid* is the DUCC-assigned process ID of the process to cancel. This is the ID in the first column of the Web Server's job details page, under the column labeled "Id".

−−**help** Prints the usage text to the console.

−−**role_administrator** The command is being issued in the role of a DUCC administrator. If the user is not also a registered administrator this flag is ignored. (This helps to protect administrators from accidentally canceling jobs they do not own.)

Notes: None.

3.7 ducc_reserve

Description: The reserve CLI is used request a reservation of resources. Reservations can be for entire machines or partial machines, based on memory requirements. All reservations are persistent: the resources remain dedicated to the requestor until explicitly returned. All reservations are performeed on an "all-or-nothing" basis: either the entire set of requested resources is reserved, or the reservation request fails.

All forms of ducc_reserve block until the reservation is complete (or fails) at which point the DUCC ID of the reservation and the names of the reserved nodes are printed to the console and the command returns.

Usage:

Script wrapper $DUCC_HOME/bin/ducc_reserve *options*

Java Main java -cp $DUCC_HOME/lib/uima-ducc-cli.jar org.apache.uima.ducc.cli.DuccReservationSubmit *options*

Options:

−−**debug** Prints internal debugging information, intended for DUCC developers or extended problem determination.

−−**description** [**text**] The text is any string used to describe the reservation. It is displayed in the Web Server.

−−**help** Prints the usage text to the console.

−−**instance_memory_size** [**integer**] This specifies the amount of memory the reserved machine must support. For full machine reservations, this is the total memory on the machine. For partial reservations, the machine may have more memory, but not less than is specified.

−−**number_of_instances** [**integer**] This specifies the number of full or partial machine reservations to schedule.

−−**scheduling_class** [**classname**] This specifies the name of the scheuling class the RM will use to determine the resource allocation for each process. The default DUCC distribution provides class "reserve" for full machine reservations, and "fixed" for partial machine reservations.

−**f**, −−**specification** [**file**] All the parameters used to request a reservation may be placed in a standard Java properties file. This file may then be used to submit the request (rather than providing all the parameters directory to submit).

Notes: Reservations may be for full machines, or partial machines based on memory. The mechanism for distinguishing which type of reservation is the job class. A job class implementing the RESERVE scheduling policy results in a full machine being reserved. A job clas simplementing the FIXED scheduling policy results in a partial machine being reserved. The default DUCC distribution configures class *reserve* for full machine reservations, and class *fixed* for partial machine reservations.

3.8 ducc_unreserve

Description: The unreserve CLI is used to release reserved resources.

Usage:

Script wrapper $DUCC_HOME/bin/ducc_unreserve *options*

Java Main java -cp $DUCC_HOME/lib/uima-ducc-cli.jar org.apache.uima.ducc.cli.DuccReservationCancel *options*

Options:

−−**debug** Prints internal debugging information, intended for DUCC developers or extended problem determination.

−−**id** [**jobid**] The ID is the id of the reservation to cancel.

−−**help** Prints the usage text to the console.

−−**role_administrator** The command is being issued in the role of a DUCC administrator. If the user is not also a registered administrator this flag is ignored. (This helps to protect administrators from inadvertently canceling jobs they do not own.)

Notes: None.

2y

3.9 ducc_process_submit

Description: Use *ducc_process_submit* to submit a Managed Reservation, also known as an *arbitrary process* to DUCC. The intention of this function is an alternative to utilities such as *ssh*, in order to allow the spawned processes to be fully managed by DUCC. This allows the DUCC scheduler to allocate the necessary resources (and prevent over-allocation), and the DUCC run-time environment to manage process lifetime.

If *attach_console* is specified, Stdin, Stderr, and Stdout of the remote process are redirected to the submitting console. It is thus possible to run interactive sessions with remote processes where the resources are managed by DUCC.

Usage:

Script wrapper $DUCC_HOME/bin/ducc_process_submit *options*

Java Main java -cp $DUCC_HOME/lib/uima-ducc-cli.jar
org.apache.uima.ducc.cli.DuccManagedReservationSubmit *options*

Options:

−−**attach_console** If specified, remote process stdout and stderr are redirected to, and stdin redirected from, the local submitting console.

−−**cancel_on_interrupt** Cancel managed reservation on interrupt (Ctrl-C). If running with −−*wait_for_completion* and this flag is specified, terminating the submit process will result in the remote process being terminated.

−−**description** [**text**] The text is any string used to describe the process. It is displayed in the Web Server. When specified on a command-line the text usually must be surrounded by quotes to protect it from the shell.

−−**debug** Prints internal debugging information, intended for DUCC developers or extended problem determination.

−−**environment** [**env vars**] Blank-delimited list of environment variable assignments for the process. Example:

```
--environment TERM=xterm DISPLAY=:1.0
```

Additional entries may be copied from the user's environment based on the setting of ducc.submit.environment.propagated in the global DUCC configuration ducc.properties.

Note: When used as a CLI option, the environment string must usually be quoted to protect it from the shell.

−−**help** Prints the usage text to the console.

−−**log_directory** [**path-to-log directory**] This specifies the path to the directory for the user logs. If not specified, the default is $HOME/ducc/logs. Example:

```
--log_directory /home/bob
```

Within this directory DUCC creates a sub-directory for each process, using the numerical ID of the job. The format of the generated log file names as described here.

Note: Note that −−log_directory specifies only the path to a directory where logs are to be stored. In order to manage multiple processes running in multiple machines DUCC, sub-directory and file names are generated by DUCC and may not be directly specified.

−−**process_executable** [**program name**] This is the full path to a program to be executed.

−−**process_executable_args** [**argument list**] This is a list of arguments for *process_executable*, if any. When specified on a command-line the text usually must be surrounded by quotes to protect it from the shell.

−−**process_memory_size** [**size**] This specifies the maximum amount of RAM in GB to be allocated to each process. This value is used by the Resource Manager to allocate resources. if this amount is exceeded by a process the Agent terminates the process with a ShareSizeExceeded message.

−−**scheduling_class** [**classname**] This specifies the name of the scheduling class the RM will use to determine the resource allocation for each process. The names of the classes are installation dependent. If not specified, the default is taken from the global DUCC configuration ducc.properties.

−−**specification,** −**f** [**file**] All the parameters used to submit a process may be placed in a standard Java properties file. This file may then be used to submit the process (rather than providing all the parameters directory to submit).

For example,

```
ducc_process_submit --specification job.props
ducc_process_submit -f job.props
```

where job.props contains:

```
working_directory   = /home/bob/projects
environment         = AE_INIT_TIME=10000 LD_LIBRARY_PATH=/a/bogus/path
log_directory       = /home/bob/ducc/logs/
```

```
description         = Simple Process
scheduling_class    = fixed
process_memory_size = 15
```

−−**suppress_console_log** If specified, suppress creation of the log files that normally hold the redirected stdout and stderr.

−−**wait_for_completion** If specified, the submit command does not return control to the console immediately, and instead monitors the DUCC state traffic and prints information about the process as it progresses.

−−**working_directory** This specifies the working directory to be set by the Job Driver and Job Process processes. If not specified, the current directory is used.

Notes:

3.10 ducc_process_cancel

Description: The cancel CLI is used to cancel a process that has previously been submitted but which has not yet completed.

Usage:

Script wrapper $DUCC_HOME/bin/ducc_process_cancel *options*

Java Main java -cp $DUCC_HOME/lib/uima-ducc-cli.jar
org.apache.uima.ducc.cli.DuccManagedReservationCancel *options*

Options:

−−**debug** Prints internal debugging information, intended for DUCC developers or extended problem determination.

−−**id [jobid]** The DUCC ID is the id of the process to cancel.

−−**help** Prints the usage text to the console.

−−**reason** Optional. This specifies the reason the process is canceled, for display in the web server.

−−**role_administrator** The command is being issued in the role of a DUCC administrator. If the user is not also a registered administrator this flag is ignored. (This helps to protect administrators from inadvertently canceling work they do not own.)

Notes: None.

3.11 ducc_services

Description: The ducc_services CLI is used to manage service registration. It has a number of functions as listed below.

The functions include:

Register This registers a service with the Service Manager by saving a service specification in the Service Manager's registration area. The specification is retained by DUCC until it is unregistered.

The registration consists primarily of a service specification, similar to a job specification. This specification is used when the Service Manager needs to start a service instance. A second properties file, the *meta properties* for the service, contains additional state and management properties. The registered properties for a service are made available for viewing from the DUCC Web Server's service details page.

Unregister This unregisters a service with the Service Manager. When a service is unregistered DUCC optionally stops the service instance, if any, and discard the saved specification.

Start The start function instructs DUCC to allocate resources for a service and to start it in those resources. The service remains running until explicitly stopped. DUCC will attempt to keep the service instances running if they should fail. The start function is also used to increase the number of running service instances if desired.

Stop The stop function stops some or all service instances.

Query The query function returns detailed information about all known services, both registered and otherwise.

Modify The modify function allows some aspects of a registered service to be updated without reregistereing the service. It optionally alters the running service instances to conform with the updates.

Usage:

Script wrapper $DUCC_HOME/bin/ducc_services *options*

Java Main java -cp $DUCC_HOME/lib/uima-ducc-cli.jar org.apache.uima.ducc.cli.DuccServiceApi *options*

The ducc_services CLI requires one of the verbs listed above as the first argument. Other arguments are determined by the verb.

Options:

3.11.1 Common Options

These options are common to all of the service verbs:

−−debug Prints internal debugging information, intended for DUCC developers or extended problem determination.

−−help Prints the usage text to the console.

3.11.2 ducc_services –register Options

The *register* function submits a service specification to DUCC. DUCC stores this information until it is *unregistered*. Once registered, a service may be started, stopped, etc.

The *register* options are logically divided into two classes:

1. "Meta" options which define how the Service Manager manages the service, and
2. Options which describe the service itself. These optinos are analagous to those used to submit a job.

The "meta" options include:

−−register [specification file] [options] The specification file is optional. If specified, it has the same contents as described for the *ducc_service_submit* command. As with *ducc_service_submit*, any of the keywords in the specification may be overridden on the command line.

−−autostart [true or false] This indicates whether to register the service as an autostarted service. If not specified, the default is *false*.

−−instances [n] This specifies the number of instances to start when the service is started. If not specified, the default is 1.

The options describing the service include:

−−classpath The CLASSPATH used for the service, if the service is a UIMA-AS services. If not specified, the CLASSPATH used by the *ducc_services* command is used.

−−classpath_order [user-before-ducc | ducc-before-user] When DUCC deploys a service, set the user-supplied CLASSPATH before DUCC-supplied CLASSPATH, or the reverse. The default is ducc-before-user. This is only valid for UIMA-AS services.

−−debug Enable debugging messages. This is primarily for debugging DUCC itself.

−−description [text] The text is any quoted string used to describe the job. It is displayed in the Web Server.

Note: When used as a CLI option, the description string must usually be quoted to protect it from the shell.

−−environment [env vars] Blank-delimited list of environment variable assignments for the service. Example:

```
--environment TERM=xterm DISPLAY=:1.0
```

Additional entries may be copied from the user's environment based on the setting of ducc.submit.environment.propagated in the global DUCC configuration ducc.properties.

Note: When used as a CLI option, the environment string must usually be quoted to protect it from the shell.

−−help This prints the usage text to the console.

−−jvm [path-to-java] This specifies the JVM to use for UIMA-AS services. If not specified, the same JVM used by the Agents is used.

Note: The path must be the full path the the Java executable (not simply the JAVA_HOME environment variable.). Example:

```
--jvm /share/jdk1.6/bin/java
```

−−process_jvm_args [list] This specifes extra JVM arguments to be provided to the server process for UIMA-AS services. It is a blank-delimited list of strings. Example:

```
--process_jvm_args -Xmx100M -Xms50M
```

Note: When used as a CLI option, the argument string must usually be quoted to protect it from the shell.

−−log_directory [path-to-log directory] This specifies the path to the directory for the individual service instance logs. If not specified, the default is $HOME/ducc/logs. Example:

```
--log_directory /home/bob
```

Within this directory DUCC creates a subdirectory for each job, using the numerical ID of the job. The format of the generated log file names as described here.

Note: Note that −−log_directory specifies only the path to a directory where logs are to be stored. In order to manage multiple processes running in multiple machines DUCC, sub-directory and file names are generated by DUCC and may not be directly specified.

−−process_DD [DD descriptor] This specifies the UIMA Deployment Descriptor for UIMA-AS services.

−−process_executable [program-name] For CUSTOM services, this specifies the full path of the program to execute.

−−process_executable_args [list-of-arguments] For CUSTOM services, this specifies the program arguments, if any.

−−process_memory_size [size] This specifies the maximum amount of RAM in GB to be allocated to each Job Process. This value is used by the Resource Manager to allocate resources.

−−instance_failures_limit [integer] This specifies the maximum number of consecutive individual service instance failures that are to be tolerated before stopping the service. The default is five (5). If the instance is successfully restarted, the count is reset to zero (0), so that the occasional process failure does not cause the entire service to be terminated.

−−scheduling_class [classname] This specifies the name of the scheuling class the RM will use to determine the resource allocation for each process. The names of the classes are installation dependent. If not specified, the default is taken from the global DUCC configuration ducc.properties.

−−service_dependency[list] This specifies a blank-delimited list of services the job processes are dependent upon. Service dependencies are discussed in detail here. Example:

```
--service_dependency UIMA-AS:Service1:tcp:node682:61616 UIMA-AS:OtherSvc:tcp:node123:123
```

Note: When used as a CLI option, the list must usually be quoted to protect it from the shell.

−−service_linger [milliseconds] This is the time in milliseconds to wait after the last referring job or service exits before stopping a non-autostarted service.

−−service_ping_arguments [argument-string] This is any arbitrary string that is passed to the *init()* method of the service pinger. The contents of the string is entirely a function of the specifici service. If not specified, a *null* is passed in.

Note: When used as a CLI option, the string must usually be quoted to protect it from the shell, if it contains blanks.

The build-in default UIMA-AS pinger supports an argument string of the following form (with NO embedded blanks):

```
queue_threshold=nn,window=mm,broker_jmx_port=pppp,meta_timeout=tttt
```

The keywords in the string have the following meaning:

queue_threshold=nn If the depth of the ActiveMQ queue for the service exceeds this value for some period of time, the health of the service is marked "poor". If omitted, the value is 0, and no quality measurement is made.

window=ww This defines a "window" over which the queue threshold is measured. the value is the number of consecutive measurements over which the queue depth must exceed the threshold. If omitted, the window size is 0.

broker_jmx_port=pppp This is the JMX port for the service's broker. If not specified, the default of 1099 is used. This is used to gather ActiveMQ statistics for the service.

meta_timeout=tttt This is the time, in milliseconds, to wait for a response to UIMA-AS *get-meta*. If not specified, the default is 5000 milliseconds.

−−service_ping_class [classname] This is the Java class used to ping a service.

This parameter is required for CUSTOM services.

This parameter may be specified for UIMA-AS services; however, DUCC supplies a default pinger for UIMA-AS services.

−−service_ping_classpath [classpath] If *service_ping_class* is specified, this is the classpath containing service_custom_ping class and dependencies. If not specified, the Agent's classpath is used (which will generally be incorrect.)

−−service_ping_dolog [boolean] If specified, write pinger stdout and stderr messages to a log, else suppress the log. See Service Pingers for details.

−−service_ping_jvm_args [string] If *service_ping_class* is specified, these are the arguments to pass to jvm when running the pinger. The arguments are specified as a blank-delimited list of strings. Example:

```
--service_ping_jvm_args -Xmx400M -Xms100M
```

Note: When used as a CLI option, the arguments must usually be quoted to protect them from the shell.

−−service_ping_timeout [time-in-ms] This is the time in milliseconds to wait for a ping to the service. If the timer expires without a response the ping is "failed". After a certain number of consecutive failed pings, the service is considered "down." See Service Pingers for more details.

−−**service_request_endpoint** **[string]** This specifies the expected service id.

This string is optional for UIMA-AS services; if specified, however, it must be of the form `UIMA-AS:queue:broker-url`, and both the queue and broker must match those specified in the service DD specifier.

If the service is CUSTOM, the endpoint is required, and must be of the form `CUSTOM:string` where the contents of the string are determined by the service.

−−**specification, −f** **[file]** All the parameters used to register a service may be placed in a standard Java properties file. This file may then be used to register the service (rather than providing all the parameters directly). For example,

```
ducc_services --register svc.props
```

where the svc.props contains:

```
environment         = AE_INIT_TIME=5000 AE_INIT_RANGE=1000 INIT_ERROR=0
description         = Test Service 3
process_jvm_args    = -DdefaultBrokerURL=tcp://agent86:61616
classpath           = /home/bob/lib/app.jar
process_memory_size = 15
working_directory   = /home/bob/services
process_DD          = /home/bob/services/service.xml
scheduling_class    = fixed
service_dependency  =  UIMA-AS:FixedSleepAE_4:tcp://agent86:61616
```

−−**working_directory** **[directory-name]** This specifies the working directory to be set for the service processes. If not specified, the current directory is used.

3.11.3 ducc_services –start Options

The start function instructs DUCC to allocate resources for a service and to start it in those resources. The service remains running until explicitly stopped. DUCC will attempt to keep the service instances running if they should fail. The start function is also used to increase the number of running service instances if desired.

−−**start** **[service-id or endpoint]** This indicates that a service is to be started. The service id is either the numeric ID assigned by DUCC when the service is registered, or the service endpoint string. Example:

```
ducc_services --start 23
ducc_services --start UIMA-AS:Service23:tcp://bob.com:12345
```

−−**instances** **[integer]** This is the number of instances to start. If omitted, sufficient instances to match the registered number are started. If more than the registered number of instances is running this command has no effect.

If the number of instances is specified, the number is added to the currently number of running instances. Thus if five instances are running and

```
          ducc_services --start 33 --instances 5
```

is issued, five more service instances are started for service 33 for a total of ten, regardless of the number specified in the registration. The registry is updated if the *–update* option is also specified. Examples:

```
ducc_services --start 23 --intances 5
ducc_services --start UIMA-AS:Service23:tcp://bob.com:12345 --instances 3 --update
```

−−**update** If specified, the registry is updated to the total number of started instances. Example:

```
ducc_services --start UIMA-AS:Service23:tcp://bob.com:12345 --instances 3 --update
```

3.11.4 ducc_services –stop Options

The stop function instructs DUCC to stop some number of service instances. If no specific number is specified, all instances are stopped. This is used only for registered services. Use *ducc_service_cancel* command to stop submitted services.

−−**stop** [**service-id or endpoint**] This specifies the service to be stopped. The service id is either the numeric ID assigned by DUCC when the service is registered, or the service endpoint string. Example:

```
ducc_services --stop 23
ducc_services --stop UIMA-AS:Service23:tcp://bob.com:12345
```

−−**instances** [**integer**] This is the number of instances to stop. If omitted, all instances for the service are stopped. If the number of instances is specified, then only the specified number of instances are stopped. Thus if ten instances are running for a service with numeric id 33 and

```
ducc_services --stop 33 --instances 5
```

is issued, five (randomly selected) service instances are stopped for service 33, leaving five running. The registry is updated if the *–update* option is specified. The registered number of instances is never reduced to zero even if the number of running instances is reduced to zero.

Example:

```
ducc_services --stop 23 --intances 5
ducc_services --stop UIMA-AS:Service23:tcp://bob.com:12345 --instances 3
```

−−**update** If specified, the registry is updated to the total number of instances remaining, but is never reduced below one (1). Example:

```
ducc_services --stop UIMA-AS:Service23:tcp://bob.com:12345 --instances 3 --update
```

3.11.5 ducc_services –modify Options

The modify function dynamically updates some of the attributes of a registered service.

−−**modify** [**service-id or endpoint**] This identifies the service to modify. The service id is either the numeric ID assigned by DUCC when the service is registered, or the service endpoint string. Example:

```
ducc_services --modify 23 --instances 3
ducc_services --modify UIMA-AS:Service23:tcp://bob.com:12345 --intances 2
```

–**instances** [**integer**] This updates the number of services instances that are started when the service is started. Only the registration is updated. If the −−*activate* option is also specified, running instances are stopped or started as needed to match the new number.

Example:

```
ducc_services --modify 23 --intances 5
ducc_services --modify UIMA-AS:Service23:tcp://bob.com:12345 --instances 3 --activate
```

–**activate** [**integer**] When specified, the number of running service instances is increased or decreased to match the newly specified number.

Example:

```
ducc_services --modify 23 --intances 5
ducc_services --modify UIMA-AS:Service23:tcp://bob.com:12345 --instances 3 --activate
```

–**autostart** [**”true” or ”false”**] This changes the autostart property for the registered services. When set to ”true”, the service is started automatically when the DUCC system is started. If the service is not currently started, it will now start with the registered number of instances. If the service is running, the instances remain running.

One way to think of this is: if *autostart* is true, DUCC will attempt to keep the registered number of instances running at all times. If *autostart* is false, all instance start and stop is manual. Example:

```
ducc_services --modify UIMA-AS:Service23:tcp://bob.com:12345 --autostart false
```

3.11.6 ducc_services –query Options

The query function returns details about all known services of all types and classes, including the DUCC ids of the service instances (for submitted and registered services), the DUCC ids of the jobs using each service, and a summary of each service's queue and performance statistics, when available.

All information returned by *ducc_services −−query* is also available via the Services Page of the Web Server as well as the DUCC Service API (see the JavaDoc).

−−**query** [**service-id or endpoint**] This indicates that a service is to be stopped. The service id is either the numeric ID assigned by DUCC when the service is registered, or the service endpoint string.

If no id is given, information about all services is returned.

Below is a sample service query.

The service with endpoint `UIMA-AS:FixedSleepAE_5:tcp://bobmach:61617` is a registered service, whose registered numeric id is 2. It was registered by bob for two instances and no autostart. Since it is not autostarted, it will be terminated when it is no longer used. It will linger for 5 seconds after the last referencing job completes, in case a subsequent job that uses it enters the system (not a realistic linger time!). It has two active instances whose DUCC Ids are 9 and 5. It is currently used (referenced) by DUCC jobs 1 and 5.

```
Service: UIMA-AS:FixedSleepAE_5:tcp://bobmach291:61617
   Service Class : Registered as ID 2 Owner[bob] instances[2] linger[5000]
   Implementors : 9 8
   References : 1 5
   Dependencies : none
   Service State : Available
   Ping Active : true
   Autostart : false
   Manual Stop : false
   Queue Statistics:
Consum Prod Qsize minNQ maxNQ expCnt inFlgt  DQ  NQ Disp
    52   44     0     0     3      0      0 402 402  402
```

Notes:

3.12 ducc_perf_stats

Description: This CLI is used to format job history and performance data into CSV or (mostly) human readable form for post-analysis. This may be run while a job is executing to monitor the current job, or after it exits. This command produces the equivalent of the web servers job details page.

Usage:

Script wrapper $DUCC_HOME/bin/ducc_perf_stats *options*

Java Main java -cp $DUCC_HOME/lib/uima-ducc-cli.jar org.apache.uima.ducc.cli.DuccPerfStats *options*

Options:

−−**job id** This specifies the job to report on.

−−**directory dir** This specifies the job's log directory. (DUCC writes usage information into this directory as the job is running.)

−−**report** [**summary or workitems or processes**] This specifies the type of report:

summary This produces a per-AE summary of the performance of that AE, including total time spent in the analytic, maximum time spent, minimum time, and total CASs processed.

workitms This produces a performance break down of each each input CAS (work item), including the work item id, ending state, time spent in queue after dispatch, processing time, the node it executed on, and the process id it ran in.

processes This produces a summary of all the processes which have executed on behalf of the job, including the node, processid, initialization time, current memory usage, maximum memory usage, page faults, swap space in use, maximum swap used, %CPU, garbage collection statistics, and work item statistics (processed, errors, retried, etc.).

−−**help** Prints the usage text to the console.

Notes: None.

3.13 viaducc and java_viaducc

Description: Viaducc is a small script wrapper around the *ducc_process_submit* CLI to facilitate launching processes on DUCC-managed machines, either from the command line or from an Eclipse run configuration.

When run from the command line as "viaducc", the arguments are bundled into the form expected by *ducc_process_submit* and submitted to DUCC. By default the remote stdin and stdout of the deployed process are mapped back to the command line terminal.

If a symbolic link to the viaducc script is created with the name "java_viaducc" and used from the command line, the arguments are assumed to be a Java classname and its arguments. The java process will be executed using DUCC's default JRE, or optionally, a specific JRE supplied by the user with a -D argument.

If the "java_viaducc" symbolic link is installed in a JRE/bin directory, DUCC will use the java executable from the same directory. More interestingly, it may be specified as an alternative to the "java" command in an eclipse launcher. The remote stdin and stdout of the deployed DUCC process are redirected to the Eclipse console. This provides essentially transparent execution of code in an Eclipse workspaces on DUCC-managed resources.

Usage:

```
viaducc [defines] [command and parameters]"
```

or

```
java_viaducc [defines] [java-class and parameters]"
```

The "defines" are described below. The "command and parameters" are either any command (with full path) and it's arguments, or a Java class (with a "main") and its arguments (including the classpath if necessary.)

Defines The arguments are specified in the syntax of Java "-D" system properties, to be more consistent with execution under Eclipse.

-DDUCC_MEMORY_SIZE This specifies the memory required, in GB. If not specified, the smallest memory quanta configured for the scheduler is used.

-DDUCC_CLASS This is the scheduling class to submit the process to. It should generally be a non-preemptable class. If not specified, it defaults to class “fixed”.

-DDUCC_ENVIRONMENT This species additional environment parameters to pass to the job. It should specify a quoted string of blank-delimeted K=V environment values. For example:

```
-DDUCC_ENVIRONMENT="DUCC_RLIMIT_NOFILE=1000 V1=V2 A=B"
```

-DJAVA_BIN This species the exact “java” command to use, for “java_viaducc”. it must be a full path to some JRE that is known to be installed on all the DUCC nodes. If not specified, the JRE used to run ducc is uses.

Chapter 4

The DUCC Public API

4.1 Overview Of The DUCC API

The DUCC API provides a simple programmatic (Java) interface to DUCC for submission and cancellation of work. (Note that the DUCC CLI is implemented using the API and provides a model for how to use the API.)

All the API objects are instantiated using the same arguments as the CLI. The API provides three variants for supplying arguments:

1. An array of Java Strings, for example `DuccJobSubmit(String[] args)`.
2. A list of Java Strings, for example `DuccJobSubmit(List<String> args)`.
3. A Java Properties object, for example `DuccJobSubmit(Properties args)`.

After instantiation of an API object, the `boolean execute()` method is called. This method transmits the arguments to DUCC. If DUCC receives and accepts the args, the method return "true", otherwise it returns "false. Methods are provided to retrieve relevant information when the `execute()` returns such as IDs, messages, etc.

In the case of jobs and managed reservations, if the specification requested debug, console attachment, or "wait for completion", the API provides methods to block waiting for completion.

In the case of jobs and managed reservations, a callback object may also be passed to the constructor. The callback object provides a means to direct messages to the API user. If the callback is not provided, messages are written to standard output.

The API is thread-safe, so developers may manage multiple, simultaneous requests to DUCC.

Below is the "main()" method of DuccJobSubmit, demonstrating the use of the API:

```
public static void main(String[] args) {
 try {
     DuccJobSubmit ds = new DuccJobSubmit(args, null);
     boolean rc = ds.execute();
     // If the return is 'true' then as best the API can tell, the submit worked
     if ( rc ) {
         System.out.println("Job " + ds.getDuccId() + " submitted");
         int exit_code = ds.getReturnCode();       // after waiting if requested
         System.exit(exit_code);
     } else {
         System.out.println("Could not submit job");
         System.exit(1);
     }
 }
```

```
        catch(Exception e) {
            System.out.println("Cannot initialize: " + e);
            System.exit(1);
        }
    }
```

4.2 Compiling and Running With the DUCC API

A single DUCC jar file is required for both compilation and execution of the DUCC API, `uima-ducc-cli.jar`. This jar is found in $DUCC_HOME/lib.

4.3 Java API

The DUCC API is documented via Javadoc in $DUCC_HOME/webserver/root/doc/apidocs/index.html.

Chapter 5

Service Management

5.1 Overview.

A DUCC service is defined by the following two criteria:

- A service is one or more long-running processes that await requests from UIMA pipeline components and return something in response. These processes are usually managed by DUCC but need not be.
- A service is accompanied by a small program called a “pinger” that the DUCC Service Manager uses to gauge the availability and health of the service. This pinger must always be be present; however, DUCC will supply a default pinger for UIMA-AS services.

A service is usually a UIMA-AS service, but DUCC supports any arbitrary process as a service.

The DUCC service manager implements several high-level functions:

- Ensure services are available for jobs before allowing the jobs to start. This is a fail-fast to prevent unnecessary allocation of resources (with potential eviction of healthy processes) for jobs that can’t run, as well as quick feedback to users that something is amiss.
- Manage the start-up and management of services: allocate resources, spawn the processes, manage the pingers, ensure the processes stay alive, handle errors, etc.
- Report on the state and availability of services.

5.2 Service Types.

DUCC supports two types of services: UIMA-AS and CUSTOM:

UIMA-AS This is a ”normal” UIMA-AS service. DUCC fully supports all aspects of UIMA-AS services with minimal effort from developers. A default “pinger” is supplied by DUCC for UIMA-AS services. (It is legal to define a CUSTOM pinger for a UIMA-AS service, however.)

CUSTOM This is any arbitrary service. Developers must provide a CUSTOM pinger and declare it in the service registration.

DUCC also supports the concept of a service that is not managed by DUCC. For example, a database or a search engine may be better managed with other facilities. In order to manage dependencies by jobs on this type of service, DUCC supports a CUSTOM service that supplies only a “pinger” and no other process. This is known as a “ping-only” service.

5.3 Service References and Endpoints

Services are referenced by an entity called a service endpoint. The service endpoint is a formatted string used to uniquely identify each service and to supply contact information to the pingers. A service endpoint is of the form

```
<service-type>:<unique id and contact information>
```

The *service-type* must be either UIMA-AS or CUSTOM.

The *unique id and contact information* is any string needed to ensure the service is uniquely named. This string is passed to the service pinger and may contain information for the pinger to contact the service. For UIMA-AS services, service endpoint is inferred by the CLI by inspection of the UIMA-AS service's DD XML descriptor. The UIMA-AS service endpoint is always of the form:

```
UIMA-AS:queue-name:broker-url
```

where *queue-name* is the name of the ActiveMQ queue used by the service, and *broker-url* is the ActiveMQ broker URL.

Jobs or other services may register dependencies on specific services by listing one or more service endpoints int their specifications. See the *job* and *services* CLI descriptions for details.

All services must be registered with DUCC. It is possible to register a "ping-only" service that has no process managed by DUCC, consisting only of a pinger.

A service is registered with DUCC using the ducc_services CLI. Service registrations are persisted by DUCC and last over DUCC and cluster restarts.

An incoming job which references a service that is not registered is marked "Services Unavailable" and canceled by the system.

There are several variants on services:

Autostarted Services An autostarted service is a service that is automatically started when the DUCC system is started. When DUCC is started, the SM checks the service registry for all service that are marked for automatic start-up. If registered for autostart, the DUCC Service Manager submits the registered number of instances on behalf of the registering user. If an instance should die, DUCC automatically restarts the instance. Short of massive failures, DUCC will ensure the service is always running and immediately ready for use with no manual intervention.

On-Demand Services An on-demand service is a registered service that is started only when referenced by the service-dependency of another job or service. If the service is already started (e.g. by reference from some other job), the dependent job/service is marked ready to schedule as indicated above. If not, the service registry is checked and if a matching service is found, DUCC starts it. When the service has completed initialization a pinger is started and all jobs waiting on it are then started.

When the last job or service that references the on-demand service exits, a timer is established to keep the service alive for a while (in anticipation that it will be needed again soon.) When the keep-alive timer expires, and there are no more dependent jobs or services, the on-demand service is automatically stopped to free up its resources for other work.

Ping-Only Services Ping-only services consist of only a ping thread. The service itself is not managed in any way by DUCC. This is useful for managing dependencies on services that are not under DUCC control: DUCC can detect and report on the health of these services and take appropriate actions on dependent jobs if the services are not responsive.

5.4 Service Pingers

A service pinger is a small program that queries a service on behalf of the DUCC Service Manager to:

- Report on the availability of the service, and

- Report on the health of the service.

Service pingers are always written in Java and must implement an abstract class,

```
org.apache.uima.ducc.common.AServicePing
```

When a service is deployed by DUCC, the Service Manager spawns a DUCC process that instantiates the pinger for the service. On a regular basis, the Service Manager sends a request to the pinger to query the service health. The pinger is expected to respond within a reasonable period of time and if it fails to do so, the service is marked unresponsive.

5.4.1 Declaring a Pinger in A Service

If your service is a UIMA-AS service, there is no need to create or declare a pinger as DUCC provides a default pinger. If a CUSTOM pinger is required, it must be declared in the service descriptor, and the service must be registered. See ducc_services for details on service registration specifying ping directives.

5.4.2 Implementing a Pinger

Pingers must implement the class `org.apache.uima.ducc.cli.AServicePing`. The abstract class is shown below.

```
import org.apache.uima.ducc.common.IServiceStatistics;

public abstract class AServicePing
{
    /**
     * Called by the ping driver, to pass in useful things the pinger may want.
     * @param arguments This is passed in from the service specification's
     *                  service_ping_arguments string.
     *
     * @param endpoint This is the name of the service endpoint, as passed in
     *                 at service registration.
     */
    public abstract void init(String arguments, String endpoint)  throws Exception;

    /**
     * Stop is called by the ping wrapper when it is being killed.  Implementors may optionally
     * override this method with conenction shutdown code.
     */
    public abstract void stop();

    /**
     * Returns the object with application-derived health and statistics.
     * @return This object contains the informaton the service manager and web server require
     *     for correct management and display of the service.
     */
    public abstract IServiceStatistics getStatistics();
}
```

Figure 5.1: Service Ping Abstract Class

Here we show the class *org.apache.uima.ducc.common.IServiceStatistics*. Custom pingers must implement this class and return an instance in response to AServicePing.getStatistics(). A default ServiceStatistics class is provide and may be used by custom pingers as *org org.apache.uima.ducc.cliServiceStatistics*. See the Javadoc for more complete details of these classes.

```
public interface  IServiceStatistics
    extends Serializable
{

    /**
     * Query whether the service is alive. This is used internally by the Service Manager.
     *
     * @return "true" if the service is responsive, "false" otherwise.
     */
    public boolean isAlive();

    /**
     * Query wether the service is "healthy". This is used internally by the Service Manager.
     * @return "true" if the service is healthy, "false" otherwise.
     */
    public boolean isHealthy();

    /**
     * Return service statistics, if any. This is used internally by the Service Manager.
     * @return A string containing information regarding the service.
     */
    public String  getInfo();

    /**
     * Set the "aliveness" of the service.  This is called by each pinger for each service.  Set
     *  this to return "true" if the service is responsive.  Otherwise return "false" so the Service
     *  Manager can reject jobs dependent on this service.
     * @param alive Set to "true" if the service is responseve, "false" otherwise.
     */
    public void setAlive(boolean alive);

    /**
     * Set the "health" of the service.  This is called by each pinger for each service.  This is a
     * subject judgement made by the service owner on behalf of his own service.  This is used only
     * to reflect status in the DUCC Web Server.
     * @param healthy Set to "true" if the service is healthy, "false" otherwise.
     */
    public void setHealthy(boolean healthy);

    /**
     * Set the monitor statistics for the service. This is any arbitray string describing critical
     * or useful characteristics of the service.  This string is presented as a "hover" in the
     * webserver over the "health" field.
     * @param info This is an arbitrary string summarizing the service's performance.
     */
    public void setInfo(String info);

}
```

Figure 5.2: ISeriveStatistics Interface

Below is a sample CUSTOM pinger for a hypothetical service that returns four integers in response to a ping.

```
import java.io.DataInputStream;
import java.io.InputStream;
import java.net.Socket;
import org.apache.uima.ducc.cli.AServicePing;
import org.apache.uima.ducc.cli.ServiceStatistics;

public class CustomPing
    extends AServicePing
{
    String host;
    String port;
    public void init(String args, String endpoint) throws Exception {
        // Parse the service endpoint, which is a String of the form
        //    host:port
        String[] parts = endpoint.split(":");
        host = parts[1];
        port = parts[2];
    }

    public void stop()  {  }

    private long readLong(DataInputStream dis) throws Exception {
        return Long.reverseBytes(dis.readLong());
    }

    public ServiceStatistics getStatistics() {
        // Contact the service, interpret the results, and return a state
        // object for the service.
        ServiceStatistics stats = new ServiceStatistics(false, false,"<NA>");
        try {
            Socket sock = new Socket(host, Integer.parseInt(port));
            DataInputStream dis = new DataInputStream(sock.getInputStream());

            long stat1 = readLong(dis); long stat2 = readLong(dis);
            long stat3 = readLong(dis); long stat4 = readLong(dis);

            stats.setAlive(true);  stats.setHealthy(true);
            stats.setInfo(  "S1[" + stat1 + "] S2[" + stat2 +
                            "] S3[" + stat3 + "] S4[" + stat4 + "]" );
        } catch ( Throwable t) {
         t.printStackTrace();
            stats.setInfo(t.getMessage());
        }
        return stats;
    }
}
```

Figure 5.3: Sample UIMA-AS Service Pinger

5.4.3 Building And Testing Your Pinger

This section provides the information needed to use the pinger API and build a custom pinger.

1. Establish compile CLASSPATH One DUCC jar is required in the CLASSPATH to build your pinger:

```
$DUCC_HOME/lib/uima-ducc-cli.jar
```

This provides the definition for the *AServicePing* and *ServiceStatistics* classes.

2. Create a registration Next, create a service registration for the pinger. While debugging, set the directive

```
service_ping_dolog = true
```

This will log any output from `System.out.println()` to your home directory in

```
$HOME/ducc/logs/services
```

Once the pinger is debugged you may want to turn logging off:

```
service_ping_dolog = false
```

A sample service registration may look something like the following:

```
bash-3.2$ cat myping.svc

description              = Ping-only service
service_request_endpoint = CUSTOM:localhost:7175
service_ping_class       = CustomPing
service_ping_classpath   = /myhome/CustomPing.class:$DUCC_HOME/lib/uima_ducc_cli.jar
service_ping_dolog       = true
service_ping_timeout     = 500
service_ping_aruments    = Arg1 Arg2
service_ping_jvm_args   = -DXmx50M
```

3. Register and start the pinger Start up your custom service so the pinger has something to contact, then start the pinger. It may be easier to debug the pinger if you initially start the service outside of DUCC. Once the pinger is working it is straightforward to integrate it into the pinger's service registration by merging the registration for the pinger with the registration for the service.

That's it! Check the web server to make sure the service "comes alive". Check your pinger's debugging log if it doesn't. Once registered, you can stop and start the pinger at will using

```
ducc_services --start <serviceid>
ducc_services --stop <serviceid>
```

where <serviceid> is the id returned when you registered the pinger.

4. If all else fails ... If your pinger does not work and you cannot determine the reason, be sure you enable *service_ping_dolog* and look in your log directory, as most problems with pingers are reflected there. As a last resort, you can inspect the the Service Manager's log in

```
$DUCC_HOME/logs/sm.log
```

Chapter 6

Job Logs

Overview The DUCC logs are managed by Apache log4j. The DUCC log4j configuration file is found in $DUCC_HOME/resources/log4j.xml. It is not in the scope of this document to describe log4j or its configuration mechanism. Details on log4j can be found at `http://logging.apache.org/log4j`.

The ”user logs” are the Job Driver (JD) and Job Process (JP) logs. There is one log for each process of a job. The JD log is divided between two physical files:

1. Stdout and default UIMA logging output written by the UIMA collection reader.
2. The diagnostic logs written by the DUCC JD wrapper around the job's collection reader.

Contents of the Log Directory A number of other useful files are written to the log directory:

1. A properties file containing the full job specification for the job. This includes all the parameters specified by the user as well as the default parameters. This file is called `job-specification.properties.`
2. The UIMA pipeline descriptor constructed by DUCC that describes the process that is dispatched to each Job Process (JP). The name of this file is of the form

   ```
   JOBID-uima-ae-descriptor-PROCESS.xml
   ```

   where

   JOBID This is the numerical id of the job as assigned by DUCC.

   PROCESS This is the process id of the Job Driver (JD) process.
3. The UIMA-AS service descriptor that defines the process that defines the job as as UIMAAS service. The name of this file is of the form

   ```
   JOBID-uima-as-dd-PROCESS.xml
   ```

   where

   JOBID This is the numerical id of the job as assigned by DUCC.

   PROCESS This is the process id of the Job Driver (JD) process.
4. A colllection of gzipped “json” files containing the performance breakdown of the job. These can be read and formatted using ducc_perf_stats.

Job Process Logs The Job Process logs are written to the configured log directory. There is one job process log for every job processes started for the job. The log names are of the following form:

```
JOBID-TYPE-NODE-PROCESS.log
```

where

JOBID This is the numerical id of the job as assigned by DUCC.

TYPE This is either the string "UIMA" for JP logs, or "JD" for JD logs.

NODE This is the name of the machine where the process ran.

PROCESS This is the Unix process id of the process on the indicated node.

Job Driver Logs There are several Job Driver logs. 988-JD-agent86-1-58087.log jd.out.log jd.err.log

Sample Log Directory This shows the contents a sample log directory for a small job that consisted of two processes.

```
100-JD-node290-1-29383.log
100-uima-ae-descriptor-29383.xml
100-uima-as-dd-29383.xml
100-UIMA-node290-2-32766.log
100-UIMA-node291-63-13594.log
jd.out.log
job-specification.properties
job-performance-summary.json.gz
job-processes-data.json.gz
work-item-status.json.gz
```

In this example,

100-JD-node290-1-29383.log
is the diagnostic JD log, where the JD executed on node node290-1 in process 29383.

100-uima-ae-descriptor-29383.xml
is the UIMA pipeline descriptor describing the service process that is launched in each JP, where the JD process is 29383.

100-uima-as-dd-29383.xml
is the UIMA-AS service descriptor where the client is the JD process running in process 29383.

100-UIMA-node290-2-32766.log
is a JP log for job 100, that ran on node node290-2, in process 32766.

100-UIMA-node291-63-13594.log
is a JP log for job 100, that ran on node node291-63, in process 13594

jd.out.log
is the user's JD log, containing the user's collection reader output.

job-performance-summary.json.gz
This contains the raw statistics describing the operation of each analytic. It corresponds to Performance tab of the Job Details page in the Web Server.

job-process.json.gz
This contains the raw statistics describing the performance of each individual job process. It corresponds Processes tab of the Job Details page in the Web Server.

work-item-status.json.gz
This contains the raw statistics describing the operation of each individual work-item. It corresponds to Work Items tab of the Job Details page in the Web Server.

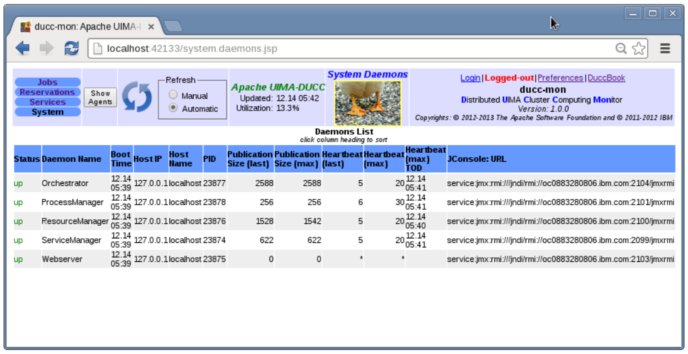
Chapter 7

DUCC Web Server

The DUCC Web Server default address is accessed from the URL http://[DUCC-HOST]:42133. The *[DUCC-HOST]* is the hostname where the local installation has installed the DUCC Web Server.

The hostname and port are configurable by the DUCC administrator in ducc.properties

The Webserver is designed to be mostly self-documenting. The design is intentionally simple and contains a link to this document. Most of the interesting fields and column headers have “mouse hovers” which display a short description if you hover your mouse pointer over it for a moment.


ducc-mon: Apache UIMA-I
localhost:42133/system.daemons.jsp
Jobs
Reservations
Services
System
Show Agents
Refresh
○ Manual
● Automatic
Apache UIMA-DUCC
Updated: 12.14 05:42
Utilization: 13.3%
System Daemons
Login|Logged-out|Preferences|DuccBook
ducc-mon
Distributed UIMA Cluster Computing Monitor
Version: 1.0.0
Copyrights: © 2012-2013 The Apache Software Foundation and © 2011-2012 IBM

Daemons List

click column heading to sort

Status	Daemon Name	Boot Time	Host IP	Host Name	PID	Publication Size (last)	Publication Size (max)	Heartbeat (last)	Heartbeat (max)	Heartbeat (max) TOD	JConsole: URL
up	Orchestrator	12.14 05:39	127.0.0.1	localhost	23877	2588	2588	5	20	12.14 05:41	service:jmx:rmi:///jndi/rmi://oc0883280806.ibm.com:2104/jmxrmi
up	ProcessManager	12.14 05:39	127.0.0.1	localhost	23878	256	256	6	30	12.14 05:41	service:jmx:rmi:///jndi/rmi://oc0883280806.ibm.com:2101/jmxrmi
up	ResourceManager	12.14 05:39	127.0.0.1	localhost	23876	1528	1542	5	20	12.14 05:40	service:jmx:rmi:///jndi/rmi://oc0883280806.ibm.com:2100/jmxrmi
up	ServiceManager	12.14 05:39	127.0.0.1	localhost	23874	622	622	5	20	12.14 05:41	service:jmx:rmi:///jndi/rmi://oc0883280806.ibm.com:2099/jmxrmi
up	Webserver	12.14 05:39	127.0.0.1	localhost	23875	0	0	*	*		service:jmx:rmi:///jndi/rmi://oc0883280806.ibm.com:2103/jmxrmi

Figure 7.1: Sample Webserver Page

Normally, the Web Server automatically fetches new data from DUCC and updates the display. This is controlled by setting one of the two refresh modes:

- Manual refresh. In this mode, the browser windows are updated only by using the browser’s refresh button, or the DUCC refresh button (to the left in the header of each page.
- Automatic refresh. In this mode, the browser automatically fetches and displays new data. The rate of refresh is currently fixed and cannot be configured.

There is a behavior difference between refresh and reload.

Refresh Refresh causes the current data on the page to be updated with the most current information in the Webserver’s possession. This is performed when the refresh button is clicked.

Reload Reload occurs when the enter key is pressed. Reload causes not just the data to be updated but rather the entire page is replaced.

Two different table styles are supported:

- Scroll, and
- Classic.

Table styles are switched using the *Preferences* link.

Scroll When *scroll* is the preference, a scroll bar is shown to the right, within the main window. The scroll bar allows scrolling to be restricted to the data display, leaving column and DUCC headers in place. In this mode any column may be sorted simply by clicking on it.

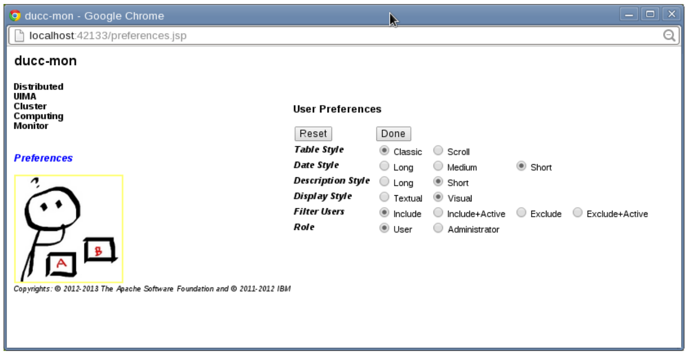
With respect to sorting, any specified sort is remembered for refresh but forgotten for reload. Sorting is permitted when either manual or automatic refresh mode is selected.

The column sort order is maintained until the page is reloaded.

Classic Mode When *classic mode* is the preference, the main data may extend below the bottom of the page and it will be necessary to use the browser's scroller on the right to access it. The column headers and DUCC header scrolls off when doing this. Columns may be sorted in this mode but it is necessary to first switch to "Manual" refresh mode to prevent browser refreshes during sorting and display of data.

With respect to sorting, any specified sort is forgotten for refresh and reload. Sorting is only permitted when manual refresh mode is selected.

The column sort order is maintained until the page is refreshed or reloaded.



User Preferences

Table Style	● Classic	○ Scroll		
Date Style	○ Long	○ Medium	● Short	
Description Style	○ Long	● Short		
Display Style	○ Textual	● Visual		
Filter Users	● Include	○ Include+Active	○ Exclude	○ Exclude+Active
Role	● User	○ Administrator		

Copyrights: © 2012-2013 The Apache Software Foundation and © 2011-2012 IBM

Figure 7.2: Preferences Page

7.1 Common Links

Every page contains a common header containing links and controls. The links permit navigation to other content at the site. The controls provide page-wise configuration of the content at that page.

The following links are available on every page of the web server:

Authentication

Authentication is needed in order to cancel jobs and reservations, to create a reservation, and to perform administration. It is not required to simply view the pages.

- Login - Authenticate and start a session with the Web Server.
- Logout - Terminate the Web Server session

Preferences The following preferences may be set:

Table Style This selects "scroll" or "classic" display, as described above.

Date Style This selects long, medium, or long formats for dates.

Description Style This selects long or short formats for the various description fields.

Filter Users This controls the "filter" box near the middle of the header on each page. It allows various levels of inclusion and exclusion of active or completed work for the filtered users.

Role This allows selection of "User" or "Administrator" roles. This protects registered DUCC administrators from accidentally affecting other people's work.

DuccBook
This is a link to the HTML version of the document you are reading.

Jobs
This navigates to the Jobs page, showing all the jobs in the system.

Reservations
This navigates to the Reservations page, showing all the reservations in the system and provides a button that can be used to request new reservations.

Services
This navigates to the Services page, showing all the services in the system.

System
This opens a sub-menu with system-related links:

- Administration - This opens a page with administrative functions.
- Classes - This shows all the scheduling classes defined to the system.
- Daemons - This shows the status of DUCC's management processes.
- DuccBook - This manual.
- Machines - This shows the status of all the ducc worker nodes.

7.2 Jobs Page

The Web Server's home page is also the Jobs page. This page has links to all the rest of the content at the site and shows the status of all the jobs in the system.

The Jobs page contains the following columns:

Id
This is the ID as assigned by DUCC. This field is hyperlinked to a Job Details page for that job that shows the breakdown of all the processes assigned to the job and their state.

Start
This is the time the Job is accepted into DUCC.

Duration
This shows two times. In green the length of time the job has been running. In black is the estimated time of completion, based on current resources and remaining work. When the job completes, the time shown is the total elapsed time of the job.

User
This is the userid of the job owner.

Class
This is the resource class the job is submitted to.

State
This shows the state of the job. The normal job progression is shown below, with an explanation of what each state means.

Received - The job has been vetted, persisted, and assigned a unique ID.

WaitingForDriver - The job is waiting for the Job Driver to initialize.

WaitingForServices - The job is waiting for verification from the Service Manager that required services are started and responding. This may cause DUCC to start services if necessary. In that even this state will persist until all pre-requisite services are ready.

WaitingForResources - The job is waiting to be scheduled. In busy systems this may require preemption of existing work. In that case this state will persist until preemption is complete.

Initializing - The job initializing. Usually this is the UIMA-AS initialization phase. In the default configuration, only two (2) processes are allocated by the Resource Manager. No additional resources are allocated until at least one of the new processes successfully completes initialization. Once initialization is complete the Resource Manager will double the number of allocated processes until the user's fair share of the resources is attained.

Running - At least one process is now initialized and running.

Completing - The last work item has completed and DUCC is freeing resources. If the job had many resources allocated at the time the job exited this state will persist until all allocated resources are freed.

Completed - The job is complete.

Reason or Extraordinary Status

This field contains miscellaneous information pertaining to the job. If the job exits the system for any reason, that reason is shown here. If the job's pre-requisite services are unavailable (or ailing) that fact is displayed here. If there is a job monitor running, that fact is shown here. Most of the values for this field support "hovers" containing additional information about the reason.

EndOfJob - The job and completed ran with no errors.

Error - All work items are processes but at least one had an error.

CanceledByDriver - The Job Driver (JD) terminated the job. The reason for termination is seen by hovering over the text with your mouse.

CanceledBySystem - The job was canceled because DUCC was shutdown.

CanceledBySser - The job owner or DUCC administrator canceled the job.

Cancel Pending - The job has been canceleled and is not yet fully evicted from the system.

DriverInitializationFailure - The Job Deiver (JD) process is unable to initialize. Hover over the field with your mouse for details (if any are available), and check your JD log.

DriverProcessFailed - The Job Driver (JD) process failed for some reason. Hover over the field with your mouse for details (if any), and check your JD log.

MonitorActive The job has a console monitor active. This is enabled with the job's "wait_for_completion" parameter on job submission.

ServicesUnavailable - The job declared a dependency on one or more services, and the Service Manager (SM) cannot find or start the required service.

Premature - The job was terminated for some unknown reason before all work items were processed. Check the JP logs for details.

ProcessInitializationFailure - Too many processes failed during initialization and the job was canceled by DUCC. Check the JP logs for the reason.

ProcessFailure - Too many processes failed while running and DUCC canceled the job. Check the JP logs for the reason.

ResourcesUnavailable - The Resource Manager (RM) is unable to allocate resources for the job. For non-preemptable jobs this could be because the limit on that type of allocation is reached, or all the nodes are already allocated and work cannot be preempted to make space for it. For all jobs, it could be because the job class is invalid.

service_name If there is a service name in this field it indicates the job is dependent on the service but the service is not responding to the Ducc Service Monitor's pinger.

Services
This is the number of services the job has declared dependencies on. There is a "hover" that shows the ids of the services, if any.

Processes
This is the number of processes currently assigned to the job.

Init Fails
This is the total number of initialization failures experienced by the job. This field is hyperlinked to pages with log excerpts highlighting the specific failures.

Run Fails
This is the total number of process failures experienced by the job. This field is hyperlinked to pages with log excerpts highlighting the specific failures.

Pgin This is the number of page-in events, over all processes, on the machines running the job.

Swap This is the total swap space, over all the processes, being used by the job.

Size
This is the declared memory size of the job

Total
This is the total number of work items declared by the job.

Done
This is the total number of work items successfully completed for the job.

Error
This is the total number of exceptions thrown or other errors experienced by work items. This field is hyperlinked to pages containing log excerpts highlighting the failures.

Dispatch
This is the total number CASs that are currently dispatched.

This usually represents the quantity derived from the following formula:

```
min( (initialized.processes * threads.per.process), (incomplete.work.items - errors) )
```

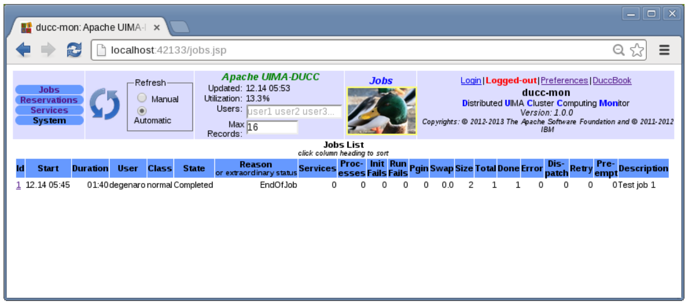
The actual number is a measured number, not a calculated number, and may differ slightly from the formula if the measurement is taken immediately after process start-up, or in the time between a work item completing and a new one being dispatched.

Retry
This is the number of CASs that were retried for any reason. Reasons for retry include preemption for fair-share, work-item timeout, or error conditions.

Note: If a work item in any process fails, the entire process is considered suspect, and all work-items in the process are terminated. Work items in the process which did not have errors are re-dispatched (retried) to a different process.

Preempt
This is the total number of processes that have been preempted to make room for other work due to Fair Share.

Description
This is the description string from the −−description string from submit.



localhost:42133/jobs.jsp

Refresh
○ Manual
● Automatic

Apache UIMA-DUCC
Updated: 12.14 05:53
Utilization: 13.3%
Users: user1 user2 user3...
Max Records: 16

Jobs

ducc-mon
Distributed UIMA Cluster Computing Monitor
Version: 1.0.0
Copyrights: © 2012-2013 The Apache Software Foundation and © 2011-2012 IBM

Jobs List
click column heading to sort

Id	Start	Duration	User	Class	State	Reason or extraordinary status	Services	Proc-esses	Init Fails	Run Fails	Pgin	Swap	Size	Total	Done	Error	Dis-patch	Retry	Pre-empt	Description
1	12.14 05:45	01:40	degenaro	normal	Completed	EndOfJob	0	0	0	0	0	0.0	2	1	1	0	0	0	0	Test job 1

Figure 7.3: Jobs Page

7.3 Job Details Page

This page shows details of all the processes that run in support of a job. The information is divided among four tabs:

Processes This tab contains details on all the processes for the job, both active, and defunct.

Work Items This tab shows details for each individual work-item in the job.

Performance This tab shows a performance break-down of all the UIMA analytics in the job.

Specification This tab shows the job specification for the job.

7.3.1 Processes

The processes page contains the following columns:

Id

This is the DUCC-assigned numeric id of the process (not the Operating System's processid). Process 0 is always the Job Driver.

Log

This is the log name for the process. It is hyperlinked to the log itself.

Size

This is the size of the log in MB. If you find you have trouble viewing the log from the Web Server it could be because it is too big to view in the server and needs to be read by some other means than the Web Server. (It is not currently paged in by the Web Server, it is read in full.)

Hostname

This is the name of the node where the process ran.

PID

This is the Unix process ID (PID) of the process.

State Scheduler

This shows the Resource Manager state of the job. It is one of:

Allocated - The node is currently allocated for this job by the RM.

Deallocated - The resource manager has deallocated the shares for the job on this node.

Reason Scheduler or extraordinary status

This column provides a reason for the scheduler state, when the scheduler state is other than "Allocated". These all have "hovers" that provide more information if it is available.

AutonomousStop - The process terminated unexpectedly of its own accord (”crashed”, or simply exited.)

Exception - The process is terminated by the JD exception handler.

Failed - The process is terminated by the Agent because the JP wrapper was able to detect and communicate a fatal condition (Exception) in the pipeline..

FailedInitialization - The process is terminated because the UIMA initialization step failed.

Forced - The node is preempted by RM for other work because of fair share.

JobCanceled - The job was canceled by the user or a system administrator.

JobCompleted - The process is canceled because of DUCC restart.

JobFailure - The job failure limit is exceeded, causing the job to be canceled by the JD.

InitializationTimeout - The UIMA initialization phase exceeded the configured timeout.

Killed - The agent terminated the process for some reason. The “Reason Agent” field should have more details in this case.

Stopped - The process was terminated by the Agent for some reason. The hover should contain more information.

Voluntary - The job is winding down, there’s no more work for this node, so it stops.

Unknown - None of the above. This is an exceptional condition, sometimes an internal DUCC error. Check the JP and JD logs for possible causes..

State Agent

This shows the DUCC Agent’s view of the state of the process.

Starting The DUCC process manager as issued a request to the assigned to start the process.

Initializing The process is initializing. Usually this means the UIMA analytic pipeline (Job Process) is executing it’s initialization method.

Running The Job Process has completed the initialization phase and is ready for, or actively executing work.

Stopped The DUCC Agent reports the process is stopped and (and has exited).

Failed The DUCC Agent reports the process failed with errors. This usually means that UIMA-AS has detected exceptions in the pipeline and reported them to the Job Driver for logging.

FailedInitialization The process died during the UIMA initialization phase.

InitializationTimeout The process exceeded the site’s limit for time spent in UIMA initialization.

Killed The DUCC Agent killed the process for some reason. There are three rosins for this:

1. The Job Processes failed to initialize,
2. The Job Process timed out during initialization,
3. The process Exocet’s its allowed swap.

Abandoned It is possible to cancel a specific process of a job. Usually this is because it became “stuck” because of hardware failure. It a process is killed in this way, the state is recoreded as *Abandonded.*

Reason Agent

This shows extended reason information if a process exited other than having run out of work to do.

AgentTimedOutWatingForORState The DUCC Agent is expecting a state update from the DUCC Orchestrator. Timer on this wait has expired. This usually indicates an infrastructure or communication problem.

Croaked The process exited for no good or clear reason, it simply vanished.

Deallocated WHAT IS THIS?

ExceededShareSize The process exceeded it's declared memory size.

ExceededSwapThreshold The process exceeded the configured swap threshold.

FailedInitialization The process was terminated because the UIMA initialization step failed.

InitializationTimeout The process was terminated because the UIMA initialization step took too long.

JPHasNoActiveJob This is set when an agent looses connectivity while its JPs are running. The job finishes (stopped or killed). The agent regains connectivity. The OR publish no longer includes the job but the agent still has processes running for that job. The agent kills ghost processes with the reason: JPHasNoActiveJob.

LowSwapSpace The process was terminated because the system is about to run out of swap space. This is a preemptive measure taken by DUCC to avoid exhaustion of swap, to effect orderly eviction of the job before the operating system starts its own reaping procedures.

AdministratorInitiated The process was canceled by an administrator.

UserInitated The process was canceled by the owning user.

Time Init
This is the clock time this process spent in initialization.

Time Run
This is the clock time this process spent in executing, not including initialization.

Time GC
This is amount of time spent in Java Garbage Collection for the process.

Count GC
This is the number of garbage collections performed by the process.

Pgin
This is the number of page-in events on behalf of the process.

Swap
This is the amount of swap space on the machine being consumed by the process.

%GC
Percentage of time spent in garbage collections by this process, relative to total of initialization + run times.

%CPU
Currant CPU percent consumed by the process. This will be > 100% on multi-core systems if more than one core is being used. Each core contributes up to 100% CPU, so, for example, on a 16-core machine, this can be as high as 1600%.

RSS
The amount of real memory being consumed by the process (Resident Storage Size)

Time Avg
This is the average time in seconds spent per work item in the process.

Time max
This is the minimum time in seconds spent per work item in the process.

Time min
This is the minimum time in seconds spent per work item in the process.

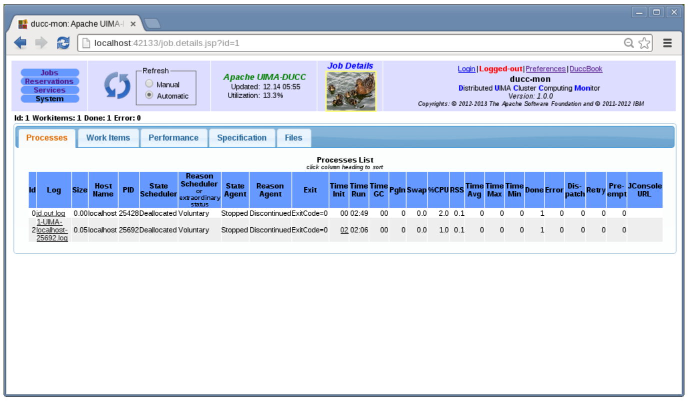
Done
This is the number of work items processed in this process.

Error
This is the number of exceptions processing work items in this process.

Retry
This is the number of work items that were retried in this process for any reason, excluding preemption.

Preempt
This is the number of work items that were preempted from this process, if fair-share caused preemption.

JConsole URL
This is a URL that can be used to connect via JMX to the processes, e.g. via jconsole.



Updated: 12.14 05:55
Utilization: 13.3%

Version: 1.0.0
Copyrights: © 2012-2013 The Apache Software Foundation and © 2011-2012 IBM

Id: 1 Workitems: 1 Done: 1 Error: 0

Processes List
click column heading to sort

Id	Log	Size	Host Name	PID	State Scheduler	Reason Scheduler or extraordinary status	State Agent	Reason Agent	Exit	Time Init	Time Run	Time GC	PgIn	Swap	%CPU	RSS	Time Avg	Time Max	Time Min	Done	Error	Dis-patch	Retry	Pre-empt	JConsole URL
0	jd.out.log	0.00	localhost	25428	Deallocated	Voluntary	Stopped	Discontinued	ExitCode=0	00	02:49	00	0	0.0	2.0	0.1	0	0	0	1	0	0	0	0	
2	1-UIMA-localhost-25692.log	0.05	localhost	25692	Deallocated	Voluntary	Stopped	Discontinued	ExitCode=0	02	02:06	00	0	0.0	1.0	0.1	0	0	0	1	0	0	0	0	

Figure 7.4: Processes Tab

7.3.2 Work Items

This tab provides details for each individual work item. Columns include:

SeqNo
This is the sequence work items are fetched from the Collection Reader's getNext() method by the DUCC Job Driver.

Id
This is the name of the work item.

Status
The is the current state of the work item. States include:

ended The work item is complete.

error The work item ended with errors.

lost The work item was queued to ActiveMQ but never dequeued by any Job Process.

operating The work item is current being executed.

retry The work item is being retried.

start The work item has been picked up for execution and DUCC is waiting for confirmation that it is running.

queued The work item has been queued to ActiveMQ but not picked up by any Job Process yet.

If a work item has not yet been retrieved from the Collect Reader it does not show on this page.

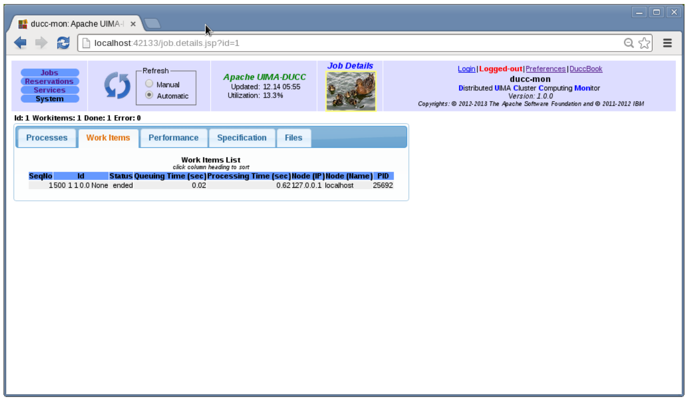
Queuing Time (sec)
The time spent in ActiveMQ after being queued, and before being picked up by a Job Process.

Processing Time (sec)
The time spent processing the work item.

Node (IP)
The node IP where the work item was processed.

Node (Name
The node name where the work item was processed.

PID
The Unix Process Id that the work item was processed in.



Apache UIMA-DUCC
Updated: 12.14 05:55
Utilization: 13.3%

ducc-mon
Distributed UIMA Cluster Computing Monitor
Version: 1.0.0
Copyrights: © 2012-2013 The Apache Software Foundation and © 2011-2012 IBM

Id: 1 Workitems: 1 Done: 1 Error: 0

Work Items List
click column heading to sort

SeqNo	Id	Status	Queuing Time (sec)	Processing Time (sec)	Node (IP)	Node (Name)	PID
1	500 1 1 0.0 None	ended	0.02	0.62	127.0.0.1	localhost	25692

Figure 7.5: Work Items Tab

7.3.3 Performance

This tab shows performance summaries of all the pipeline components. The statistics are aggregated over all instances of each component in each process of the job.

Name
The short name of the analytic. The full name is shown in the command-line tool ducc_perf_stats

Total
This is the total time in days, hours, minutes, and seconds taken by each component of the pipeline.

% of Total
This is the percent of the total usage consumed by this analytic.

Avg
This is the average time spent by all the instances of the analytic.

Min
This is the minimum time spent by any instance of the analytic.

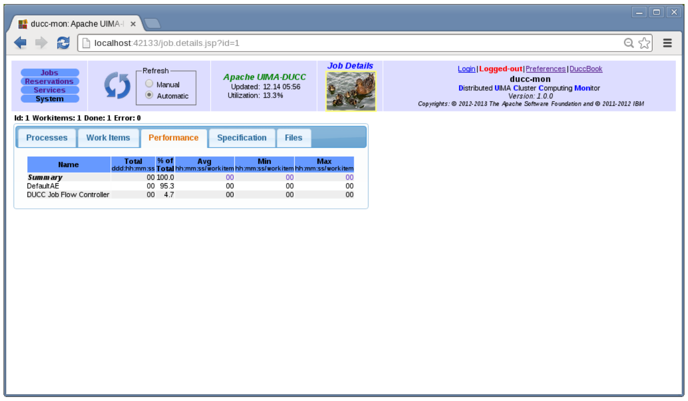
Max
This is the maximum time spent by any instance of the analytic.

7.3.4 Specification

This tab shows the full job specification in the form of a Java Properties file. This will include all the parameters specified by the user, plus those filled in by DUCC.

7.4 Reservation Page

This page shows details of all reservations. There are two types of reservations: *managed* and *unmanaged*..



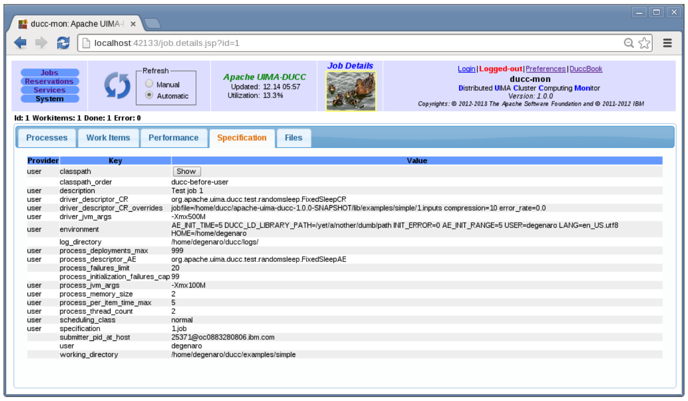
Apache UIMA-DUCC
Updated: 12.14 05:56
Utilization: 13.3%

ducc-mon
Distributed UIMA Cluster Computing Monitor
Version: 1.0.0
Copyrights: © 2012-2013 The Apache Software Foundation and © 2011-2012 IBM

Id: 1 Workitems: 1 Done: 1 Error: 0

Name	Total ddd:hh:mm:ss	% of Total	Avg hh:mm:ss/workitem	Min hh:mm:ss/workitem	Max hh:mm:ss/workitem
Summary	00	100.0	00	00	00
DefaultAE	00	95.3	00	00	00
DUCC Job Flow Controller	00	4.7	00	00	00

Figure 7.6: Performance Tab



Apache UIMA-DUCC
Updated: 12.14 05:57
Utilization: 13.3%

ducc-mon
Distributed UIMA Cluster Computing Monitor
Version: 1.0.0
Copyrights: © 2012-2013 The Apache Software Foundation and © 2011-2012 IBM

Id: 1 Workitems: 1 Done: 1 Error: 0

Provider	Key	Value
user	classpath	Show
	classpath_order	ducc-before-user
user	description	Test job 1
user	driver_descriptor_CR	org.apache.uima.ducc.test.randomsleep.FixedSleepCR
user	driver_descriptor_CR_overrides	jobfile=/home/ducc/apache-uima-ducc-1.0.0-SNAPSHOT/lib/examples/simple/1.inputs compression=10 error_rate=0.0
user	driver_jvm_args	-Xmx500M
user	environment	AE_INIT_TIME=5 DUCC_LD_LIBRARY_PATH=/yet/a/nother/dumb/path INIT_ERROR=0 AE_INIT_RANGE=5 USER=degenaro LANG=en_US.utf8 HOME=/home/degenaro
	log_directory	/home/degenaro/ducc/logs/
user	process_deployments_max	999
user	process_descriptor_AE	org.apache.uima.ducc.test.randomsleep.FixedSleepAE
	process_failures_limit	20
	process_initialization_failures_cap	99
user	process_jvm_args	-Xmx100M
user	process_memory_size	2
user	process_per_item_time_max	5
user	process_thread_count	2
user	scheduling_class	normal
user	specification	1.job
	submitter_pid_at_host	25371@oc0883280806.ibm.com
	user	degenaro
	working_directory	/home/degenaro/ducc/examples/simple

Figure 7.7: Specification Tab

A *managed reservation* is a reservation whose process is fully managed by DUCC. This process is any arbitrary process and is submitted with the ducc_process_submit CLI. The lifetime of the reservation starts at the time DUCC assigns a unique ID, and ends when the process terminates for any reason.

An *unmanaged reservation* is essentially a sandbox for the user. DUCC starts no processes in the reservation and manages none of the processes which run on that node. The lifetime of the reservation starts at the time DUCC assigns a unique ID, and ends when the submitter or system administrator cancels it. *Managed reservations* can potentially last an indefinite period of time.

The Reservations page contains the following columns:

Id

This is the unique DUCC numeric id of the reservation as assigned when the reservation is made. If this is a *managed* reservation, the ID is hyperlinked to a Managed Reservation Details page with extended details on the process running in the reservation.

Start

This is the time the reservation was mode.

End

This is the time the reservation was canceled or otherwise ended.

User

This is the userid if the person who made the reservation.

Class

This is the scheduling class used to schedule the reservation.

Type

This is the reservation type, *managed* or *unmanaged*, as described above.

State

This is the status of the reservation. Values include: Received - Reservation has been vetted, persisted, and assigned unique Id.

Assigned - The reservation is active.

Completed - The reservation has been terminated.

Received - The Reservation has been vetted, persisted, and assigned a unique ID.

WaitingForResources - The reservation is waiting for the Resource Manager to find and schedule resources.

Reason

If a reservation is not active, this shows the reason. Note that for *unmanaged reservations*, even if the user has processes running in the reservation, DUCC does NOT attempt to terminate those processes (hence, "unmanaged".)

For *managed reservations*, DUCC does terminate the associated process.

CanceledBySystem - In the case of the special JobDriver reservation, this is canceled by DUCC and reestablished on reboot; hence the state is a result of DUCC having been restarted.

In all other cases, it is a result of DUCC being restarted *COLD*. When DUCC is started *COLD*, all previous reservations are canceled. (When DUCC is started *WARM*, the default, previous reservations are preserved.)

CanceledByAdmin - The DUCC administrator released the reservation.

CanceledByUser - The reservation owner released the reservation.

ResourcesUnavailable - The Resource Manager was unable to find free or freeable resources match the resource request.

ProgramExit - The reservation is a *managed* reservation and the associated process has exited.

Allocation

This is the number of resources (shares for FIXED policy reservations, processes for RESERVE policy reservations) that are allocated.

UserProcesses This is the number of processes owned by the user running in all shares of the reservation.

Note that even for *unmanaged* reservations, the DUCC agent tracks processes owned by the user and reports on them. This allows better identification and management of abandoned reservations.

Size

The memory size in GB of the each allocated unit. This is the amount of memory that was *requested*. In the case of RESERVE policy reservations, that actual memory of the reserved machine may be greater.

Host Names

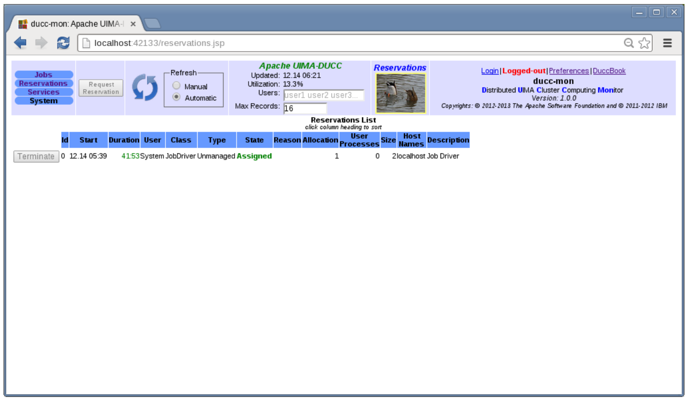
The node names of the machines where the resources are allocated.

Description

This is the description string from the –description string from submit.

7.5 Managed Reservation Details Page

This page shows details of the processes which run in a managed reservation. The information is divided between two tabs:



localhost:42133/reservations.jsp

Apache UIMA-DUCC
Updated: 12.14 06:21
Utilization: 13.3%
Users: user1 user2 user3...
Max Records: 16

ducc-mon
Distributed UIMA Cluster Computing Monitor
Version: 1.0.0
Copyrights: © 2012-2013 The Apache Software Foundation and © 2011-2012 IBM

Reservations List
click column heading to sort

	Id	Start	Duration	User	Class	Type	State	Reason	Allocation	User Processes	Size	Host Names	Description
Terminate	0	12.14 05:39	41:53	System	JobDriver	Unmanaged	Assigned		1	0	2	localhost	Job Driver

Figure 7.8: Reservations Page

Processes This tab contains details on all the proceses contained in the reserved space.

Specification This tab shows the specification for the process.

7.5.1 Processes

The processes page contains the following columns:

ID

This is the DUCC-assigned numeric id of the process. This format of this id is two numbers:

```
RESID.SHAREID
```

Here, the *RESID* is the reservation ID. The *SHAREID* is the share ID assigned by the Resource Manager. Together these form a unique ID for each process that runs in the reservation.

Note: The current version of DUCC supports only one process per managed reservation. Future versions are expected to support multiple processes within a single managed reservation.

Log

This is the log name for the process. It is hyperlinked to the log itself.

Size

This is the size of the log in MB. If you find you have trouble viewing the log from the web server it could be because it is too big to view in the browser.

Hostname

This is the name of the node where the process is running (or ran).

PID

This is the Unix process ID (PID) of the process.

State Scheduler

This shows the Resesource Manager state of the job. It is one of:

Allocated - The node is still allocated for this job by the RM.

Deallocated - The resource manager has deallocated the shares for the job on this node.

Reason Scheduler or Extraordinary Status

These are the same as for the job details.

State Agent

These are the same as for the job details.

Reason Agent
These are the same as for the job details.

Time Run
The current duration of the reservation, or total duration if it has terminated.

RSS
The amount of real memory being consumed by the process (Resident Storage Size)

7.5.2 Specification

This tab shows the full job specification in the form of a Java Properties file. This will include all the parameters specified by the user, plus those filled in by DUCC.

7.6 Services Page

This page shows details of all services.

The Services page contains the following columns:

Id
This is the unique numeric DUCC id of the service. This ID is hyperlinked to a Service Details page with extended details on the service. Note that for some types of services, DUCC may not know more about the service than is shown on the main page.

Name
This is the unique service endpoint of the service.

Type
This is the service type, which is one of

- Registered
- Ping-Only

State
This is the state of the service with respect to the service manager. It is a consolidated state over all the service instances. Valid states are

Available At least one service instance is responding to the service pinger, indicating it is functional.

Initializing No service instances are available for use yet but at least one instance is in its UIMA *initializing* phase.

Waiting At least one service instance is in Running state, potentially available for use, but no response has been received from the service pinger. This usually occurs during the start-up of a service. If a service stops responding to its pinger after becoming available, the state can regress to Waiting.

NotAvailable No service instance is running or initializing.

Stopping The service has been stopped for some reason, but not all instances have terminated. This is an intermediate state between Available and NotAvailable to signify that the service is no longer available but not all its resources have been returned yet.

DUCC will start dependent jobs ONLY if it's services are in state Available. Otherwise DUCC attempts to start the service, and if successful, allows the job to start.

If a job is already running and a service becomes other than Available, the jobs page indicates the service is not available but the job is allowed to continue.

Pinger

This indicates whether the Service Manager is running a pinger for the service. This column does not imply the ping is successful; see the "health" column for ping status.

Health

Health is a status returned by each pinger and is the result of that pinger's evaluation of the state of the service. It is shown as on of

- *Good*
- *Poor*

Both terms are highly subjective. Pingers may return a summary of the underlying data used to label a service as good or bad. That status is shown as a hover over this field.

Instances

This is the number of instances (processes) currently registered for the service.

Deployments

This is the number of actual instances deployed for the service. Note that this may be greater, or less, than the number of registered instances, if the service owner decides to temporarily start or stop additional instances.

User

This is the userid of the service owner.

Class

This is the scheduling class the service is running in.

If a service is registered as "ping-only", no resources are allocated for it. This is shown as a class of `ping-only`.

Size

This is the memory size, in GB, of each service instance

Jobs

This is the number of jobs currently using the service. The IDs of the jobs are shown as hovers over this field.

Services

Services may themselves depend on other services. This field shows the number of services dependent on this service. The dependent service IDs are shown with a hover over the field.

Reservations

This field shows the number of managed reservations dependent on this service. The IDs of the managed reservations are shown as a hover over the field.

Description

This is the description string from the –description string from submit.

7.7 Service Details Page

This page shows details of the processes which implement.

The information is divided between two tabs:

Processes This tab contains details on all the proceses implementing the service, if any.

Specification This tab shows the specification for the service.

7.7.1 Processes

The processes page contains the following columns:

ID

This is the DUCC-assigned numeric id of the process. This format of this id is two numbers:

```
RESID.SHAREID
```

Here, the *RESID* is the reservation ID. The *SHAREID* is the share ID assigned by the Resource Manager. Together these form a unique ID for each process that runs in the reservation.

Log

This is the log name for the process. It is hyperlinked to the log itself.

Size

This is the size of the log in MB. If you find you have trouble viewing the log from the web server it could be because it is too big to view in the browser.

Hostname

This is the name of the node where the process is running (or ran).

PID

This is the Unix process ID (PID) of the process.

State Scheduler

This shows the Resesource Manager state of the job. It is one of:

Allocated - The node is still allocated for this job by the RM.

Deallocated - The resource manager has deallocated the shares for the job on this node.

Reason Scheduler or Extraordinary Status

These are the same as for the job details.

State Agent

These are the same as for the job details.

Reason Agent

These are the same as for the job details.

Time Init

Most services are UIMA-AS services and therefore have an *initialization* phase to their lifetimes. This field shows the time spent in that phase.

Time Run

The current duration of the reservation, or total duration if it has terminated.

Time GC

This is amount of time spent in Java Garbage Collection for the process.

Pgin

This is the number of page-in events on behalf of the process.

Swap

This is the amount of swap space on the machine being consumed by the process.

%CPU

Currnt CPU percent consumed by the process. This will be > 100% on multi-core systems if more than one core is being used. Each core contributes up to 100% CPU, so, for example, on a 16-core machine, this can be as high as 1600%.

RSS

The amount of real memory being consumed by the process (Resident Storage Size)

JConsole URL

This is a URL that can be used to connect via JMX to the processes, e.g. via jconsole.

7.7.2 Specification

This tab shows the full job specification in the form of a Java Properties file. This will include all the parameters specified by the user, plus those filled in by DUCC.

The specification for a Service contains two types of entries:

1. Service specification properties, prefixed with "svc". These comprise the service specification that the Service Manager submits on behalf of a user in order to start registered services.
2. Meta properties, prefixed with "meta". This is the Service Manager's state record for the sesrvice as it is running. In addition to state it contains properties required for service registration that are not used for service submission.

7.8 System Details Page

This page shows information relating to the DUCC System itself:

Administration This displays system administrators and implements the interface to various administrative controls.

Classes This shows the current system's scheduling class definitions.

Daemons This shows the status of all DUCC processes.

DuccBook This is a link to the book you are reading.

Machines This shows details of all the machines in the DUCC cluster.

7.8.1 Administration

This page has two tabs:

Administrators This shows the user-ids that are authorized to administer DUCC. In addition to executing the "Control" functions described below, administrators may cancel any job, reservation, or service, and may modify services they do not own.

In order to perform administrative functions, the following must be satisfied:

1. The user is logged-in to the web server.
2. The user is a registered administrator.
3. The user has set the role as "administrator" in the DUCC Preferences page. This is a safeguard so that administrators who are also users are less likely to inadvertently affect other people's jobs.

Control Currently DUCC supports a single administrative control function via the web server: Stop new job submissions and re-enable them. If submissions are blocked, all existing work runs normally, but no new work is accepted.

7.8.2 Classes

This page shows the definitions of the DUCC scheduling classes. The scheduling classes are discussed in more detail in the Resource Manager section.

7.8.3 Daemons

This page shows the current state of all DUCC processes. By default, only the administrative processes, Orchestrator, ProcessManager, ResourceManager, ServiceManager, and Webserver are shown. A button in the upper left of the page titled "Show Agents" enables display of the status of all the DUCC agents as well. (Agents are suppressed by default because the page is expensive to render for large systems.)

The columns shown on this page include

Status

This indicates whether the daemon is running and broadcasting state *up*, or not *down*.

All DUCC daemons broadcast a heartbeat containing process state. If the Status is *down*, either the daemon is not functioning, or something is preventing state from reaching the web server via DUCC's ActiveMQ instance.

Daemon Name

This is the name of the process.

Boot Time

This shows the date and time of the latest boot of the specific process.

Host IP

This is the IP address of the processor where the process is running.

Host Name

This shows the hostname of the processor where the process is running.

PID

This is the Unix processid of the DUCC process.

Publication Size (last)

This shows the size of the most recent state publication of the process, in bytes.

Publication Size (max)

This shows the size of the largest state publication of the process, in bytes.

Heartbeat (last)

This shows the number of seconds since the last state publication for the process. Large numbers here indicate potential cluster or DUCC problems.

Heartbeat (max)

This shows the longest delay since a state publication for the process was received at the web server. Large numbers here indicate potential cluster or DUCC problems.

Heartbeat (max) TOD

This shows the time the longest delay of a state publication occurred.

JConsole URL

This is the jconsole URL for the process.

7.8.4 Machines

This page shows the states of all the machines in the DUCC cluster.

The columns shown on this page include

Status

This shows the current state of a machine. Values include:

defined The node is in the DUCC nodes file, but no DUCC process has been started there, or else there is a communication problem and the state messages are not being delivered.

up The node has a DUCC Agent process running on it and the web server is receiving regular heartbeat packets from it.

down The node had a healthy DUCC Agent on it at some point in the past (since the last DUCC boot), but the web server has stopped receiving heartbeats from it.

The agent may have been manually shut down, may have crashed, or there may be a communication problem.

Additionally, very heavy loads from jobs running the the node can cause the DUCC Agents heartbeats to be delayed.

IP

This is the IP address of the node.

Name

This is the hostname of the node.

Reserve(GB) size

This is the largest reservation that can be made on this node.

This is usually somewhat less than the physical memory size because it is rounded down to the nearest share quantum. The purpose of this column is to assist users in requesting the right size for full machine reservations.

Memory(GB) total

This is the amount of memory, in GB, as reported by each machine.

Usually the amount will be slightly less than the installed memory. This is because a small bit of memory is usually reserved by the hardware for its own purposes. For example, a machine with 48GB of installed memory may report only 47GB available.

Swap(GB) in use

This is the total size in-use swap data. DUCC shows any value greater than 0 in red as swapping can very significantly slow applications. However, swap use does not always mean there is a performance problem. This is flagged by DUCC simply as an alert of a potential problem

Alien PIDs

This shows the number of processes not owned by DUCC, the operating system, or jobs scheduled on each node. The Unix Process IDS of these processes is displayed in a hover.

DUCC preconfigures many of the standard operating system process and userids. This list may be updated by each installation.

A common cause of alien PIDs is errant process run in unmanaged reservations. A user may reserve a machine for use as a sandbox. If the reservation is released without properly terminating all the processes, they may linger. When ducc schedules the node for other purposes, significant performance penalties may be paid due to competition between the legitimately scheduled work and the leftover “alien” processes. The purpose of this column is to bring attention to this situation.

Shares (total)

This shows the total number of scheduling share supported on this node.

Shares(in use)

This shows the total number of scheduling share in use on the node.

Heartbeat(last)

This shows the number of seconds since the last agent heartbeat from this machine.

Part III

Programming Model And Applications

Chapter 8

Building and Testing Jobs

8.1 Overview

A DUCC job consists of two process types, a Job Driver process and one or more Job Processes. These processes are connected together via UIMA-AS. The Job Driver process wraps the job's Collection Reader (CR). The CR function is to define the collection of Work Items to be processed. The Collection Reader returns a small CAS for each Work Item containing a reference to the Work Item data The Job Driver uses the UIMA-AS client API to send Work Item CASes to the job's input queue. Job Processes containing the analytic pipeline are deployed as UIMA-AS services and comsume CASes from the job input queue.

A basic job's analytic pipeline consists of an Aggregate Analysis Engine comprised by the user specified CAS Multiplier (CM), Analysis Engine (AE) and CAS Consumer (CC) components, along with a built-in DUCC Flow Controller. The Work Item CAS is typically sent only to the CM and returned by the Job Process when all child CASes produced by the CM have completed processing; optionally the CR can configure Work Item CAS flow to go to the CC or to the AE & CC to complete all processing for that Work Item.

Note: Although the Job Driver will receive back the Work Item CAS, there is no provision for any user code to receive the CAS. Therefore a Job Process typically adds no results to a Work Item CAS.

8.1.1 Basic Job Process Threading Model

In addition to the pipeline definition of explicitly named CM, AE and CC components, the job specification also includes the number of pipeline threads to run in each Job Process (using the job specification parameter: process_thread_count). Each pipeline thread receives Work Items independently.

DUCC creates an aggreate descriptor for the pipeline, and then creates a Deployment Descriptor for the Job Process which specifying the number of synchronous pipelines.

8.1.2 Alternate Pipeline Threading Model

Alternately a Job Process can be fully specified by a user submitted UIMA-AS Deployment Descriptor. Thus any UIMA-AS service deployment can be used as a Job Process. Here the parameter process_thread_count just defines how many Work Items CASes will be sent to each Job Process concurrently.

Note: In general a UIMA-AS service may be configured to return child CASes; although child CASes returned from a Job Process will be ignored by the Job Driver, there may be significant overhead in wasted serialization and I/O.

8.1.3 Overriding UIMA Configuration Parameters

UIMA configuration parameters in the CR, CM, AE or CC components can be overriden using job specification parameters: driver_descriptor_CR_overrides, process_descriptor_CM_overrides, process_descriptor_AE_overrides and process_descriptor_CC_overrides, respectively.

Another approach is to use the *External Configuration Parameter Overrides* mechanism in core UIMA. External overrides is the only approach available for jobs submitted with a Deployment Descriptor.

8.2 Collection Segmentation and Artifact Extraction

UIMA is built around artifact processing. A classic UIMA pipeline starts with a Collection Reader (CR) that defines collection seqmentation, extracts the artifacts to be analyzed and puts them into the CASes to be delivered to subsequent analytic components. A CR designed for a specific data collection is highly reusable for many different analytic scenarios.

A single CR supplying artifacts to a large number of analysis pipelines would be a bottleneck. Not only would artifact data need to be transported twice across the compute cluster, but analysis results would be uselessly returned to the Job Driver. To solve both of these problems, in a DUCC job the CR only sends a reference to the artifacts in the Work Item CAS, and artifact data is read directly by the analysis pipeline.

In DUCC collection processing the role of collection segmentation is implemented by the CR run in the Job Driver, while artifact extraction and CAS initialization are implemented in the Cas Multiplier (CM) run in the Job Process. The combination of a CR and associated CM should be highly reusable.

Note: In many cases it is useful to reference multiple artifacts in a Work Item CAS. Both DUCC sample applications described below exhibit this design.

8.3 CAS Consumer Changes for DUCC

CAS Consumers in a UIMA pipeline may require changes for scale out into DUCC jobs, to avoid scale out bottlenecks, to preserve collection level processing, or to flush results at end-of-work-item processing.

Federated ouput: Scaled out DUCC jobs distribute artifact processing to multiple pipeline instances. All instances of a CAS Consumer should have independent access to the output target (filesystem, service, database, etc.).

Singleton processing: Collection level processing requiring that all results go to a singleton process would usually be done as a follow-on job, allowing incremental progress; Job Process errors due to data-dependent analysis bugs can often be fixed without invalidating completed Work Items, enabling a restarted job to utilize the progress made by previous job runs.

Flushing cached data: In some scenarios each Work Item delivered to a pipeline can be considered an independent collection. If a CAS Consumer caches data which needs to be flushed after processing the last artifact for a Work Item, the Work Item CAS can be routed to the CAS Consumer after the last artifact CAS is processed and used to trigger cache flushing.

8.4 Job Development for an Existing Pipeline Design

Assuming that an existing job input-output design (CR, CM, CC) is to be reused, job development is focussed on the Analysis Engine (AE) to be plugged in. Before deploying a new AE in a multithreaded Job Process it is best to run it single threaded (process_thread_count=1) to separate basic logic errors from threading problems.

To debug a Job Process with eclipse, first create a debug configuration for a ”remote java application”, specifying ”Connection Type = Socket Listen” on some free port P. Start the debug configuration and confirm it is listening on the specified port. Then add to the job specification process_debug=port, where port is the value P used in the running debug configuration.

When the process_debug parameter is specified, DUCC will only run a single Job Process that will connect back to the eclipse debug configuration.

8.5 Job Development for a New Pipeline Design

A DUCC job is a UIMA application comprised of user code broken into a Collection Reader running in the Job Driver and an Agreggate Analysis Engine (analysis pipeline) running in one or more Job Processes. Each Job Process may run multiple instances of the pipeline, each in a different thread. The major components of the basic Job Process application are as follows:

- User Collection reader - segments the input collection in to Work Items
- User CAS Multiplier - inputs a Work Item and segments it into artifacts (CASes)
- User Analysis Engine - processes the CASes
- User CAS Consumer - outputs results for each Work Item
- DUCC built-in Flow Controller - routes Work Item CASes to the CM and optionally to the CC or AE & CC.

It is good if the CR+CM+CC combination can be reused for a broad range of AE.

8.5.1 DUCC built-in Flow Controller

This flow controller provides separate flows for Work Item CASes and for CASes produced by the CM and/or AE. Its behavior is controlled by the existence of a CM component, and then further specified by the org.apache.uima.ducc.Workitem feature structure in the Work Item CAS.

When no CM is defined the Work Item CAS is simply delivered to the AE, and then to the CC if defined. Any CASes created by the AE will be routed to the CC.

With a defined CM, the Work Item CAS is delivered only to the CM, and then returned from the JP when processing of all child CASes created by the CM and AE has completed. Work Item CAS flow can be further refined by the CR by creating a org.apache.uima.ducc.Workitem feature structure and setting the setSendToLast feature to true, or by setting the setSendToAll feature to true.

8.5.2 Workitem Feature Structure

In addition to Work Item CAS flow control features, the WorkItem feature structure includes other features that are useful for a DUCC job application. Here is the complete list of features:

sendToLast (Boolean) - indicates the Work Item CAS be sent to the CC

sendToAll (Boolean) - indicates Work Item CAS be sent to the AE and CC

inputspec (String) - reference to Work Item input data

outputspec (String) - reference to Work Item output data

encoding (String) - useful for reading Work Item input data

language (String) - used by the CM for setting document text language

bytelength (Integer) - size of Work Item

blockindex (Integer) - used if a Work Item is one of multiple pieces of an input resource

blocksize (Integer) - used to indicate block size for splitting an input resource

lastBlock (Boolean) - indicates this is the last block of an input resource

8.5.3 Deployment Descriptor (DD) Jobs

Job Processes with arbitrary aggregate hierarchy, flow control and threading can be fully specified via a UIMA AS Deployment Descriptor. DUCC will modify the input queue to use DUCC's private broker and change the queue name to correspond to the DUCC job ID.

8.5.4 Debugging

It is best to develop and debug the interactions between job application components as one, single-threaded UIMA aggregate. DUCC provides an easy way to accomplish this, for both basic and DD job models, using the all_in_one specification parameter.

all_in_one=local When set to local, all Job components are run in the same single-threaded process, on the same machine as eclipse.

all_in_one=remote With remote, the single-threaded process is run on a DUCC worker machine as a DUCC Managed Reservation.

To debug an all_in_one job with eclipse, first create a debug configuration for a "remote java application", specifying "Connection Type = Socket Listen" on some free port P. Start the debug configuration and confirm it is listening on the specified port. Then, before submitting the all_in_one job, add the argument process_debug=port, where port is the value P used in the running debug configuration.

Chapter 9

Sample Application: Raw Text Processing

9.1 Application Function and Design

This application expects as input a directory containing one or more flat text files, uses paragraph boundaries to segment the text into separate artifacts, processes each artifact with the OpenNlpTextAnalyzer, and writes the results as compressed UIMA CASes packaged in zip files. Paragraph boundaries are defined as two or more consecutive newline characters.

By default each input file is a Work Item. In order to facilitate processing scale out, an optional blocksize parameter can be specified that will be used to break larger files into multiple Work Items. Paragraphs that cross block boundaries are processed in the block where they started. An error is thrown if a paragraph crosses two block boundaries.

An output zip file is created for each Work Item. The CAS compression format is selectable as either ZIP compressed XmiCas or UIMA compressed binary form 6 format. When compressed binary is used, each zip file also contains the full UIMA Type System in ZIP compressed text. CASes in UIMA compressed binary form 6 format have the same flexibility as an XmiCas in that they can be deserialized into a CAS with a different, but compatible Type System.

By default any previously completed output files found in the output directory are preserved. While Work Item processing is in progress the associated output files have "_temp" appended to their filenames, and any such incomplete output files are always ignored for subsequent jobs.

9.2 Configuration Parameters

The Collection Reader for this job is the DuccJobTextCR. It has the following configuration parameters:

InputDirectory path to directory containing input files.

OutputDirectory path to directory for output files.

IgnorePreviousOutput (optional) boolean to ignore (overwrite) existing output files.

Encoding (optional) character encoding of the input files.

Language (optional) language of the input documents, i.e. cas.setDocumentLanguage(language).

BlockSize (optional) integer value used to break larger input files into multiple Work Items.

SendToLast (optional) boolean to route WorkItem CAS to last pipeline component. Is set to true for this application.

SendToAll (optional) boolean to route WorkItem CAS to all pipeline components. Not used in this application.

The CAS Consumer is the DuccCasCC and has the following configuration parameters:

XmiCompressionLevel (optional) compression value if using ZIP compression. Default is 7, range is 0-9.

UseBinaryCompression (optional) boolean to select UIMA binary CAS compression.

9.3 Set up a working directory

For this and the following sample program, create a working directory in a writable filesystem.

Copy to this directory the example job specification files:

```
cp $DUCC_HOME/examples/sampleapps/descriptors/*.job .
```

Copy a UIMA logger configuration file that suppresses tons of output from OpenNLP:

```
cp $DUCC_HOME/examples/sampleapps/descriptors/ConsoleLogger.properties .
```

Copy the executable code and resources for the DUCC sample application components:

```
mkdir lib
cp $DUCC_HOME/lib/uima-ducc/examples/uima-ducc-examples*.jar lib
```

For reference the source code for DUCC sample applications is in $DUCC_HOME/examples/src, with descriptors in $DUCC_HOME/examples/sampleapps/descriptors.

9.4 Download and Install OpenNLP

Download the OpenNLP source distribution from http://opennlp.apache.org and follow the directions in the *UIMA Integration* section of the included documentation to build the UIMA pear file. Then *install* the UIMA pear file in the working directory with the *runPearInstaller* script and test it with the UIMA Cas Visual Debugger application.

A small modification of the installed OpenNLP descriptor file is necessary for DUCC to run the component multithreaded. Edit *opennlp.uima.OpenNlpTextAnalyzer/desc/OpenNlpTextAnalyzer.xml* and change the setting for *multipleDeploymentAllowed* from false to true.

9.5 Get some Input Text

Choose one or more flat text files in UTF8 format that only use newline characters, *not CR-LF sequences.* The text should be big enough to see the impact of DUCC job scale out. We used test data from gutenberg.org at

```
http://www.gutenberg.org/ebooks/search/?sort_order=downloads
```

downloading 'Plain Text UTF-8' versions of *Moby Dick*, *War and Peace* and *The Complete Works of William Shakespeare* as flat text files in a subdirectory 'Books', and removing all 'CR' characters (0xD) as well as extraneous text.

9.6 Run the Job

The job specification, DuccRawTextSpec.job, uses placeholders to reference the working directory and various operational components located there. As run below the placeholders are resolved from environmental variables.

The job is submitted from the command line with the following:

```
MyAppDir=$PWD \
MyInputDir=$PWD/Books \
MyOutputDir=$PWD/Books.processed \
$DUCC_HOME/bin/ducc_submit -f DuccRawTextSpec.job
```

The total size of the three txt files is 9.4Mbytes and with a blocksize of 100000 there are 100 Work Items. Each Job Process is configured to run 8 parallel OpenNLP pipelines. To examine the performance of processing with just a single Job Process, the job can be submitted as:

```
MyAppDir=$PWD \
MyInputDir=$PWD/Books \
MyOutputDir=$PWD/Books.processed \
$DUCC_HOME/bin/ducc_submit -f DuccRawTextSpec.job \
--process_deployments_max 1
```

9.7 Job Output

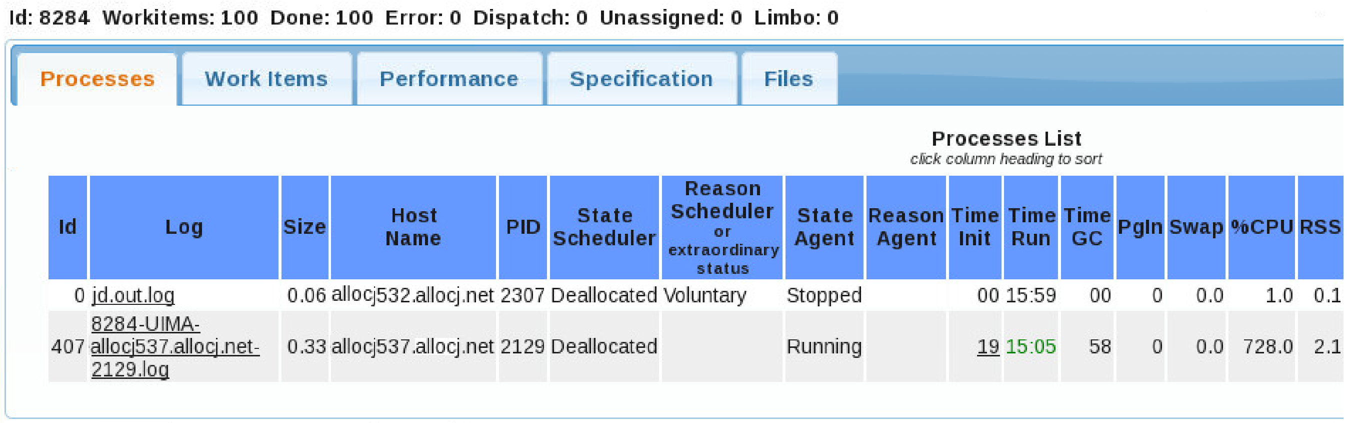
There will be an output zipfile for every Work Item, with zipfiles containing a compressed CAS for each document (paragraph) found in a Work Item. If UseBinaryCompression=true each zipfile will also contain the TypeSystem for the CASes. This is needed when deserializing these CASes into a different TypeSystem.

DuccTextCM finds 19245 paragraphs in the three txt files. If the output CASes are stored as 19245 uncompressed XMI files, the total size is 911MB. Using the default ZIP compressed XMI format and packed into 100 Work Item zip files, the total size is 165MB, a 5.5x compression. Using UIMA binary compressed format further reduces total size to 62MB.

This output data will be used as input data for the following CAS input processing sample application.

9.8 Job Performance Details

DUCC captures a number of process performance metrics. Figure 9.1 shows details on the JD and single JP processes. The %CPU time shown, 728, is lower than the actual because the Job Process was idle for some time before it received the first Work Item and also idle between finishing the last Work Item and being shut down. DUCC shows the JVM spent a total of 58 seconds in GC (garbage collection), had no major page faults or page space, and used a max of 2.1GB of RSS.



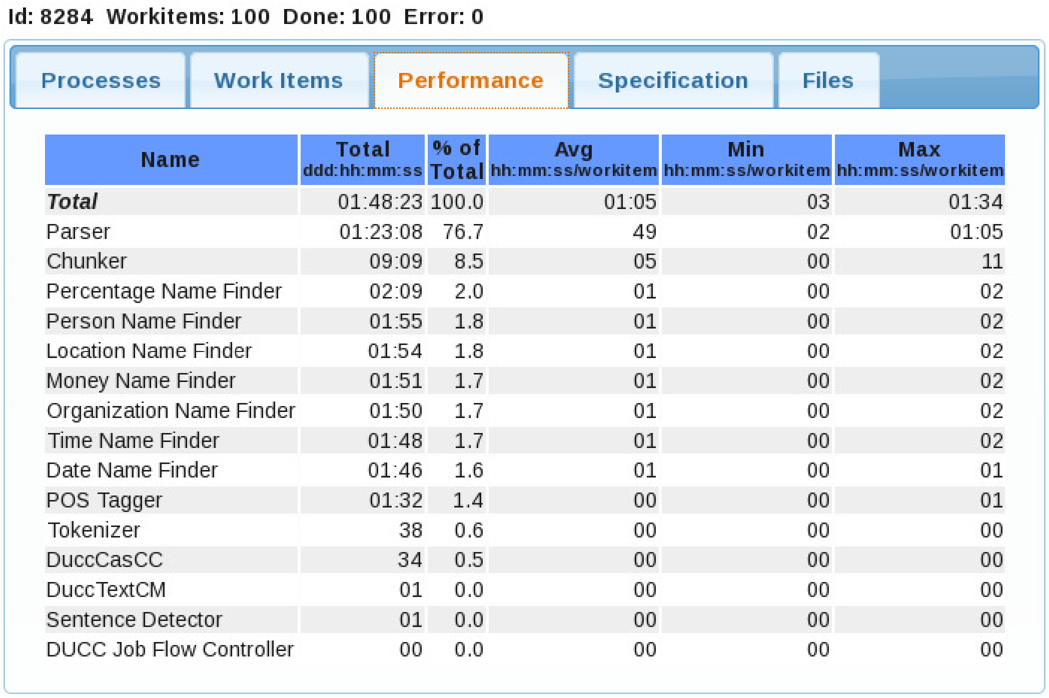
Id: 8284 Workitems: 100 Done: 100 Error: 0 Dispatch: 0 Unassigned: 0 Limbo: 0

Processes List
click column heading to sort

Id	Log	Size	Host Name	PID	State Scheduler	Reason Scheduler or extraordinary status	State Agent	Reason Agent	Time Init	Time Run	Time GC	PgIn	Swap	%CPU	RSS
0	jd.out.log	0.06	allocj532.allocj.net	2307	Deallocated	Voluntary	Stopped		00	15:59	00	0	0.0	1.0	0.1
407	8284-UIMA-allocj537.allocj.net-2129.log	0.33	allocj537.allocj.net	2129	Deallocated		Running		19	15:05	58	0	0.0	728.0	2.1

Figure 9.1: OpenNLP Process Measurements

On the Performance tab, DUCC shows the breakdown of clock time spent in each primitive UIMA component running in the Job Process. See Figure 9.2. Processing time was dominated by the Parser component at 76.7%. The time spent compressing and writing out CASes was 0.5%, and the time reading the input text files well below 0.1%.



Id: 8284 Workitems: 100 Done: 100 Error: 0

Name	Total ddd:hh:mm:ss	% of Total	Avg hh:mm:ss/workitem	Min hh:mm:ss/workitem	Max hh:mm:ss/workitem
Total	01:48:23	100.0	01:05	03	01:34
Parser	01:23:08	76.7	49	02	01:05
Chunker	09:09	8.5	05	00	11
Percentage Name Finder	02:09	2.0	01	00	02
Person Name Finder	01:55	1.8	01	00	02
Location Name Finder	01:54	1.8	01	00	02
Money Name Finder	01:51	1.7	01	00	02
Organization Name Finder	01:50	1.7	01	00	02
Time Name Finder	01:48	1.7	01	00	02
Date Name Finder	01:46	1.6	01	00	01
POS Tagger	01:32	1.4	00	00	01
Tokenizer	38	0.6	00	00	00
DuccCasCC	34	0.5	00	00	00
DuccTextCM	01	0.0	00	00	00
Sentence Detector	01	0.0	00	00	00
DUCC Job Flow Controller	00	0.0	00	00	00

Figure 9.2: OpenNLP Process Breakdown

Chapter 10

Sample Application: CAS Input Processing

10.1 Application Function and Design

The main purpose of this application is to demonstrate the overhead of processing a collection of CASes grouped into zipfiles and stored as ZIP compressed XmiCas or with UIMA compressed binary form 6 format.

Note: This application depends on successful processing of the work in the previous chapter.

10.2 Configuration Parameters

The Collection Reader for this job is the DuccJobCasCR. It has the following configuration parameters:

InputSpec path to directory containing input files (named InputSpec in the hope that more options will be added).

OutputDirectory path to directory for output files.

IgnorePreviousOutput (optional) boolean to ignore (overwrite) previous output files.

SendToLast (optional) boolean to route WorkItem CAS to last pipeline component. Set to true in this application.

SendToAll (optional) boolean to route WorkItem CAS to all pipeline components. Not used in this application.

The CAS Consumer is the DuccCasCC and has the following configuration parameters:

XmiCompressionLevel (optional) compression value if using ZIP compression. Default is 7.

UseBinaryCompression (optional) boolean to select UIMA binary CAS compression.

10.3 Run the Job

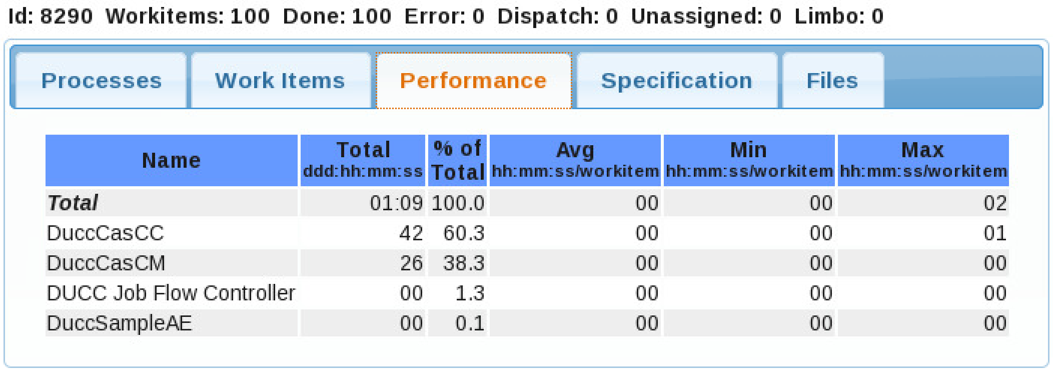
The job specification, DuccCasInputSpec.job, uses placeholders to reference the working directory and various operational components located there. As run below the placeholders will be resolved from environmental variables.

The job is submitted from the command line with the following:

```
MyAppDir=$PWD \
MyInputDir=$PWD/Books.processed \
MyOutputDir=$PWD/Books.followon \
$DUCC_HOME/bin/ducc_submit -f DuccCasInputSpec.job \
--process_deployments_max 1
```

10.4 Job Performance Details

Figure 10.1 shows the component breakdown using binary CAS compression. Reading and deserializing took 38% vs the 60% spent serializing and writing. Using 8 pipeline threads in one process the 19245 CASes output from the last application were read and re-written in 9 seconds.



Id: 8290 Workitems: 100 Done: 100 Error: 0 Dispatch: 0 Unassigned: 0 Limbo: 0

Name	Total ddd:hh:mm:ss	% of Total	Avg hh:mm:ss/workitem	Min hh:mm:ss/workitem	Max hh:mm:ss/workitem
Total	01:09	100.0	00	00	02
DuccCasCC	42	60.3	00	00	01
DuccCasCM	26	38.3	00	00	00
DUCC Job Flow Controller	00	1.3	00	00	00
DuccSampleAE	00	0.1	00	00	00

Figure 10.1: CAS Input Processing Performacne

10.5 Limiting Job Resources

Although this 8-threaded Job Process was primarily CPU bound doing serialization work, it is possible to become I/O bound with enough threads banging on a shared filesystem. DuccCasInputSpec.job demonstrates how to limit the total number of processing threads to 32 using the combination of process_thread_count=8 and process_deployments_max=4.

I/O vs CPU bottlenecks can be detected using the detailed performance job data reported by DUCC and comparing results with various levels of scale out.

Part IV

Ducc Administrators Guide

Chapter 11

Installation, Configuration, and Verification

11.1 Overview

DUCC is a multi-user, multi-system distributed application. First-time installation is performed in two stages:

- Single-user installation: This provides single-user, single-system installation for testing, and verification. Simple development of small applications on small systems such as laptops or office workstations is possible after Single-user Installation.
- Multi-user installation: This provides secure multi-user capabilities and configuration for multi-system clusters.

First-time users must perform single-user installation and verification on a single system. Once this configuration is working and verified, it is straightforward to upgrade to a multi-user configuration.

Note: A key feature of DUCC is to run user processes in CGroups in order to guarantee each process always has the amount of RAM requested. Total RAM allocation to the managed process and its child processes that exceed requested memory will go to swap space. Without CGroups a process that exceeds its requested memory size by N% is killed (default N=5 in ducc.properties), and memory use by child processes is ignored.

DUCC is distributed as a compressed tar file. The instructions below assume installation from one of this distribution media. If building from source, the build creates this file in your svn trunk/target directory. The distribution file is in the form

```
uima-ducc-[version]-bin.tar.gz
```

where [version] is the DUCC version; for example, *uima-ducc-1.0.0-bin.tar.gz.* This document will refer to the distribution file as the "<distribution.file>".

11.2 Software Prerequisites

Both single and multi-user configurations have the following software pre-requisites:

- A userid *ducc*, and group *ducc*. User *ducc* must the the only member of group *ducc*.
- Reasonably current Linux. DUCC has been tested on SLES 10.2, 11.1, and 11.2, and RHEL 6.3.

  Note: On some systems the default *user limits* for max user processes (ulimit -u) and nfiles (ulimit -n) are defined too low for DUCC. The shell login profile for user *ducc* should set the soft limit for max user processes to be the same as the hard limit (ulimit -u 'ulimit -Hu'), and the nfiles limit raised above 1024 to at least twice the number of user processes running on the cluster.

- For CGroupt support, libcgroup1-0.37+ on SLES and libcgroup-0.37+ on RHEL.
- DUCC has been tested and run on IBM and Oracle JDK 1.6 and 1.7.
- Python 2.x, where 'x' is 4 or greater. DUCC has not been tested on Python 3.x.

Multi-user installation has additional requirements:

- All systems must have a shared filesystem (such as NFS or GPFS) and common user space
- Passwordless ssh must be installed for user *ducc* on all systems.
- Root access is required to install a small setuid-root program on each system.

In order to build DUCC from source the following software is also required:

- A Subversion client, from `http://subversion.apache.org/packages.html`. The svn url is `https://svn.apache.org/repos/asf/uima/sandbox/uima-ducc/trunk`.
- Apache Maven, from `http://maven.apache.org/index.html`

The DUCC webserver server optionally supports direct "jconsole" attach to DUCC job processes. To install this, the following is required:

- Apache Ant, any reasonably current version.

To (optionally) build the documentation, the following is also required:

- Latex, including the *pdflatex* and *htlatex* packages. A good place to start if you need to install it is `https://www.tug.org/texlive/`.

More detailed one-time setup instructions for source-level builds via subversion can be found here: `http://uima.apache.org/one-time-setup.html#svn-setup`

11.3 Building from Source

To build from source, ensure you have Subversion and Maven installed. Extract the source from the SVN repository named above.

Then from your extract directory into the root directory (usually current-directory¿/trunk), and run the command

```
mvn install
```

or

```
mvn install -Pbuild-duccdocs
```

if you have LaTeX insalled and wish to do the optional build of documentation.

If this is your first Maven build it may take quite a while as Maven downloads all the open-source pre-requisites. (The pre-requisites are stored in the Maven repository, usually your $HOME/.m2).

When build is complete, a tarball is placed in your current-directory/trunk/target directory.

11.4 Documentation

After single-user installation, the DUCC documentation is found (in both PDF and HTML format) in the directory ducc_runtime/docs. As well, the DUCC webserver contains a link to the full documentation on each major page. The API is documented only via JavaDoc, distributed in the webserver's root directory `$DUCC_HOME/webserver/root/doc/apidocs.`

If building from source, Maven places the documentation in

- `trunk/uima-ducc-duccdocs/target/site` (main documentation), and

- `trunk/site/apidocs` (API Javadoc)

11.5 Single-user Installation and Verification

Single-user installation sets up an initial, working configuration on a single system. Although any user ID can be used to run DUCC in single-user mode, it is recommended to create user "ducc" to avoid confusion. No security is established in single-user mode, and all user processes run as the DUCC user ID.

Verification submits a very simple UIMA pipeline for execution under DUCC. Once this is shown to be working, one may proceed to upgrade to full installation.

11.6 Minimal Hardware Requirements for single-user Installation

- One Intel-based or IBM Power-based system. (More systems may be added during multi-user installation, described below.)
- 8GB of memory. 16GB or more is preferable for developing and testing applications beyond the non-trivial.
- 1GB disk space to hold the DUCC runtime, system logs, and job logs. More is usually needed for larger installations.

Please note: DUCC is intended for scaling out memory-intensive UIMA applications over computing clusters consisting of multiple nodes with large (16GB-256GB or more) memory. The minimal requirements are for initial test and evaluation purposes, but will not be sufficient to run actual workloads.

11.7 Single-user System Installation

1. Expand the distribution file:

   ```
   tar -zxf <distribution.file>
   ```

   This creates a directory with a name of the form "apache-uima-ducc-[version]".

   This directory contains the full DUCC runtime which you may use "in place" but it is highly recommended that you move it into a standard location; for example, under ducc's HOME directory:

   ```
   mv apache-uima-ducc-1.0.0 /home/ducc/ducc_runtime
   ```

   We refer to this directory, regardless of its location, as $DUCC_HOME. For simplicity, some of the examples in this document assume it has been moved to /home/ducc/ducc_runtime.

2. Change directories into the admin sub-directory of $DUCC_HOME:

   ```
   cd $DUCC_HOME/admin
   ```

3. Run the post-installation script:

   ```
   ./ducc_post_install
   ```

   If this script fails, correct any problems it identifies and run it again.

   Note that *ducc_post_install* initializes various default parameters which may be changed later by the system administrator. Therefore it usually should be run only during this first installation step.

4. If you wish to install jconsole support from the webserver, make sure Apache Ant is installed, and run

   ```
   ./sign_jconsole_jar
   ```

   This step may be run at any time if you wish to defer it.

That's it, DUCC is installed and ready to run. (If errors were displayed during ducc_post_install they must be corrected before continuing.)

The post-installation script performs these tasks:

1. Verifies that the correct level of Java and Python are installed and available.
2. Creates a default nodelist, $DUCC_HOME/resources/ducc.nodes, containing the name of the node you are installing on.
3. Defines the "ducc head" node to be to node you are installing from.
4. Sets up the default https keystore for the webserver.
5. Installs the DUCC documentation "ducc book" into the DUCC webserver root.
6. Builds and installs the C program, "ducc_ling", into the default location.
7. Ensures that the (supplied) ActiveMQ broker is runnable.

11.8 Initial System Verification

Here we start the basic installation, submit a simple UIMA-AS job, verify that it ran, and stop DUCC. Once this is confirmed working DUCC is ready to use in an unsecured, single-user mode on a single system.

To run the verification, issue these commands.

1. cd $DUCC_HOME/admin
2. ./check_ducc -s

   Examine the output of check_ducc. If any errors are shown, correct the errors and rerun check_ducc until there are no errors.
3. Finally, start ducc: ./start_ducc -s

Start_ducc will first perform a number of consistency checks. It then starts the ActiveMQ broker, the DUCC control processes, and a single DUCC agent on the local node. Note that "single user mode" (-s) is specified for this first start. This inhibits the checks on the permissions on ducc_ling (described below).

You will see some startup messages similar to the following:

```
ENV: Java is configured as: /share/jdk1.6/bin/java
ENV: java full version "1.6.0_14-b08"
ENV: Threading enabled: True
MEM: memory is 15 gB
ENV: system is Linux
allnodes /home/ducc/ducc_runtime/resources/ducc.nodes
Class definition file is ducc.mixed.classes
OK: Class and node definitions validated.
OK: Class configuration checked
Starting broker on ducchead.biz.org
Waiting for broker ..... 0
Waiting for broker ..... 1
ActiveMQ broker is found on configured host and port: ducchead.biz.org:61618
Starting 1 agents
********** Starting agents from file /home/ducc/ducc_runtime/resources/ducc.nodes
Starting warm
Waiting for Completion
ducchead.biz.org Starting rm
     PID 14198
ducchead.biz.org Starting pm
```

```
     PID 14223
ducchead.biz.org Starting sm
     PID 14248
ducchead.biz.org Starting or
     PID 14275
ducchead.biz.org Starting ws
     PID 14300
ducchead.biz.org
    ducc_ling OK
    DUCC Agent started PID 14325
All threads returned
```

Now open a browser and go to the DUCC webserver's url, http://<hostname>:42133 where <hostname> is the name of the host where DUCC is started. Navigate to the Reservations page via the links in the upper-left corner. You should see the DUCC JobDriver reservation in state WaitingForResources. In a few minutes this should change to Assigned. (This usually takes 3-4 minutes in the default configuration.) Now jobs can be submitted.

The hostname and port are configurable by the DUCC administrator in ducc.properties

To submit a job,

1. $DUCC_HOME/bin/ducc_submit –specification $DUCC_HOME/examples/simple/1.job

Open the browser in the DUCC jobs page. You should see the job progress through a series of transitions: Waiting For Driver, Waiting For Services, Waiting For Resources, Initializing, and finally, Running. You'll see the number of work items submitted (15) and the number of work items completed grow from 0 to 15. Finally, the job will move into Completing and then Completed..

Since this example does not specify a log directory DUCC will create a log directory in your HOME directory under

```
$HOME/ducc/logs/job-id
```

In this directory, you will find a log for the sample job's JobDriver (JD), JobProcess (JP), and a number of other files relating to the job.

This is a good time to explore the DUCC web pages. Notice that the job id is a link to a set of pages with details about the execution of the job.

Notice also, in the upper-right corner is a link to the full DUCC documentation, the "DuccBook".

Finally, stop DUCC:

1. cd $DUCC_HOME/admin
2. ./stop_ducc -a

Once the system is verified and the sample job completes correctly, proceed to Multi-User Installation and Verification to set up multiple-user support and optionally, multi-node operation.

11.9 Logs

The DUCC system logs are written to the directory

```
    $DUCC_HOME/logs
```

In that directory are found logs for each of the DUCC components plus one for each node DUCC is installed on.

DUCC job/user logs are written by default to the user's HOME directory under

```
    $HOME/ducc/logs/<jobid>
```

11.10 Multi-User Installation and Verification

Multi-user installation consists of these steps over and above single-user installation:

1. Install and configure the setuid-root program, ducc_ling. This small program allows DUCC jobs to be run as the submitting user rather than user ducc.
2. Optionally install and configure CGroups.
3. Optionally update the configuration to include additional nodes.

Multi-user installation has these pre-requisites (DUCC will not work on multiple nodes unless these steps are taken):

- All systems in the DUCC cluster must have a shared filesystem and shared user space (user directories are shared over NFS or an equivalent networked file system, across the systems, and user ids and credentials are the same).
- Passwordless ssh must be installed for user ducc on all systems. Users do NOT require ssh access to the DUCC nodes.
- Root access is (briefly) required to install a small setuid-root program on each system.

11.11 Ducc_ling Installation

Before proceeding with this step, please note:

- This step is required ONLY to install multi-user capabilities.
- The sequence operations consisting of *chown* and *chmod* MUST be performed in the exact order given below. If the *chmod* operation is performed before the *chown* operation, Linux will regress the permissions granted by *chmod* and ducc_ling will be incorrectly installed.

ducc_ling is a setuid-root program whose function is to execute user tasks under the identity of the user. This must be installed correctly; incorrect installation can prevent jobs from running as their submitters, and in the worse case, can introduce security problems into the system.

ducc_ling must be installed on LOCAL disk on every system in the DUCC cluster, to avoid shared-filesystem access to it. The path to ducc_ling must be the same on each system. For example, one could install it to /local/ducc/bin on local disk on every system.

To install ducc_ling (these instructions assume it is installed into /local/ducc/bin): As ducc, ensure ducc_ling is built correctly for your architecture:

1. cd $DUCC_HOME/duccling/src
2. make clean all

Now, as root, move ducc_ling to a secure location and grant authorization to run tasks under different users' identities:

1. mkdir /local/ducc
2. mkdir /local/ducc/bin
3. chown ducc.ducc /local/ducc
4. chown ducc.ducc /local/ducc/bin
5. chmod 700 /local/ducc
6. chmod 700 /local/ducc/bin
7. cp $DUCC_HOME/duccling/src/ducc_ling /local/ducc/bin
8. chown root.ducc /local/ducc/bin/ducc_ling

9. chmod 4750 /local/ducc/bin/ducc_ling

Finally, update the configuration to use this ducc_ling instead of the default ducc_ling:

1. Edit $DUCC_HOME/resources/ducc.properties and change this line:

   ```
    ducc.agent.launcher.ducc_spawn_path=${DUCC_HOME}/admin/ducc_ling
   ```

   to this line (Using the actual location of the updated ducc_ling, if different from /local/ducc/bin):

   ```
   ducc.agent.launcher.ducc_spawn_path=/local/ducc/bin/ducc_ling
   ```

What these steps do:

1. The first two step compile ducc_ling for your current machine architecture and operating system level.
2. The next two steps (mkdir) create directory /local/ducc/bin
3. The next two steps (chown) set ownership of /local/ducc and /local/ducc/bin to user ducc, group ducc
4. The next two steps (chmod) set permissions for /local/ducc and /local/ducc/bin so only user ducc may access the contents of these directories
5. The copy stop copies the ducc_ling program created in initial installation into /local/ducc/bin
6. The next step (chown) sets ownership of /local/ducc/bin/ducc_ling to root, and group ownership to ducc.
7. The next step (chmod) stablishes the *setuid* bit, which allows user ducc to execute ducc_ling with root privileges.
8. Finally, ducc.properties is updated to point to the new, privileged ducc_ling.

If these steps are correctly performed, ONLY user *ducc* may use the ducc_ling program in a privileged way. Ducc_ling contains checks to prevent even user *root* from using it for privileged operations.

Ducc_ling contains the following functions, which the security-conscious may verify by examining the source in $DUCC_HOME/duccling. All sensitive operations are performed only AFTER switching userids, to prevent unauthorized root access to the system.

- Changes it's real and effective userid to that of the user invoking the job.
- Optionally redirects its stdout and stderr to the DUCC log for the current job.
- Optionally redirects its stdio to a port set by the CLI, when a job is submitted.
- "Nice"s itself to a "worse" priority than the default, to reduce the chances that a runaway DUCC job could monopolize a system.
- Optionally sets user limits.
- Prints the effective limits for a job to both the user's log, and the DUCC agent's log.
- Changes to the user's working directory, as specified by the job.
- Optionally establishes LD_LIBRARY_PATH for the job from the environment variable

  ```
          DUCC_LD_LIBRARY_PATH
  ```

  if set in the DUCC job specification. (Secure Linux systems will prevent LD_LIBRARY_PATH from being set by a program with root authority, so this is done AFTER changing userids).

11.12 CGroups Installation and Configuration

The steps in this task must be done as user root and user ducc.

To install and configure CGroups for DUCC:

1. Install the appropriate libcgroup package at level 0.37 or above.

2. Configure /etc/cgconfig.conf as follows:

```
# Mount cgroups
mount {
   memory = /cgroup;
}
# Define cgroup ducc and setup permissions
group ducc {
 perm {
     task {
        uid = ducc;
     }
     admin {
        uid = ducc;
     }
 }
 memory {}
}
```

3. Start the cgconfig service:

```
service cgconfig start
```

4. Verify cgconfig service is running by the existence of directory:

```
/cgroups/ducc
```

5. Some versions of libcgroup are buggy. As user ducc, verify that the following command executes with no error:

```
cgcreate -t ducc -a ducc -g memory:ducc/test-cgroups
```

6. Configure the cgconfig service to start on reboot:

```
chkconfig cgconfig on
```

7. Update ducc.properties to enable CGroups. Note that if CGroups is not installed, the DUCC Agent will detect this and not attempt to use the feature. It is completely safe to enable CGroups in *ducc.properties* on machines where it is not installed.

11.13 Set up the full nodelists

To add additional nodes to the ducc cluster, DUCC needs to know what nodes to start its Agent processes on. These nodes are listed in the file

```
$DUCC_HOME/resources/ducc.nodes.
```

During initial installation, this file was initialized with the node DUCC is installed on. Additional nodes may be added to the file using a text editor to increase the size of the DUCC cluster.

11.14 Full DUCC Verification

This is identical to initial verification, with the one difference that the job “1.job” should be submitted as any user other than ducc. Watch the webserver and check that the job executes under the correct identity. Once this completes, DUCC is installed and verified.

11.15 Enable DUCC webserver login

This step is optional. As shipped, the webserver is disabled for logins. This can be seen by hovering over the Login text located in the upper right of most webserver pages:

```
System is configured to disallow logins
```

To enable logins, a Java-based authenticator must be plugged-in and the login feature must be enabled in the ducc.properties file by the DUCC administrator. Also, ducc_ling should be properly deployed (see Ducc_ling Installation section above).

A beta version of a Linux-based authentication plug-in is shipped with DUCC. It can be found in the source tree:

```
org.apache.uima.ducc.ws.authentication.LinuxAuthenticationManager
```

The Linux-based authentication plug-in will attempt to validate webserver login requests by appealing to the host OS. The user who wishes to login provides a userid and password to the webserver via https, which in-turn are handed-off to the OS for a success/failure reply.

To have the webserver employ the beta Linux-based authentication plug-in, the DUCC administrator should perform the following as user ducc:

```
1. edit ducc.properties
2. locate: ducc.ws.login.enabled = false
3. modify: ducc.ws.login.enabled = true
4. save
```

Note: The beta Linux-based authentication plug-in has limited testing. In particular, it was tested using:

```
Red Hat Enterprise Linux Workstation release 6.4 (Santiago)
```

Alternatively, you can provide your own authentication plug-in. To do so:

```
1. author a Java class that implements
   org.apache.uima.ducc.common.authentication.IAuthenticationManager
2. create a jar file comprising your authentication class
3. put the jar file in a location accessible by the DUCC webserver, such as
    $DUCC_HOME/lib/authentication
4. put any authentication dependency jar files there as well
5. edit ducc.properties
6. add the following:
   ducc.local.jars = authentication/*
   ducc.authentication.implementer=<your.authenticator.class.Name>
7. locate: ducc.ws.login.enabled = false
8. modify: ducc.ws.login.enabled = true
9. save
```

Chapter 12

Administration

12.1 WebServer Authentication

By default, DUCC is configured such that there is effectively no authentication enforcement by the WebServer. No password entry is permitted on the Login panel and any userid specified is accepted whether it exists or not.

To enable your own authentication measures, you should perform the following steps:

1. Author an authentication manager Java class implementing interface

   `org.apache.uima.ducc.common.authentication.IAuthenticationManager`

2. Create an authentication jar file comprising the authentication manager Java class
3. Install your authentication jar file and any dependency jar files into your DUCC's lib folder
4. Update your ducc.properties file with authentication class name and jar file name(s) information
5. Create a ducc.administrators file

Note: When a user clicks on the WebServer Login link, the login dialog is shown. On that dialog panel is shown the authenticator: *version*, which is supplied by your authentication manager implementation's *getVersion()* method. Also shown are boxes for userid and password entry. If your authentication manager implemenation's *isPasswordChecked()* method returns true then the password box will accept input, otherwise it will be disabled.

12.1.1 Example Implementation

Shown below is an example implementation which can be used as a template for coding protection by means of interfacing with your site's security measures.

In this example, the SiteSecurity Java class is presumed to be existing and available code at your installation.

```
package org.apache.uima.ducc.example.authentication.module;

import org.apache.uima.ducc.common.authentication.AuthenticationResult;
import org.apache.uima.ducc.common.authentication.IAuthenticationManager;
import org.apache.uima.ducc.common.authentication.IAuthenticationResult;
import org.apache.uima.ducc.example.authentication.site.SiteSecurity;

public class AuthenticationManager implements IAuthenticationManager {

    private final String version = "example 1.0";
    @Override
    public String getVersion() {
        return version;
    }

    @Override
    public boolean isPasswordChecked() {
        return true;
    }

    @Override
    public IAuthenticationResult isAuthenticate(String userid, String domain,
            String password) {
        IAuthenticationResult authenticationResult = new AuthenticationResult();
        authenticationResult.setFailure();
        try {
            if(SiteSecurity.isAuthenticUser(userid, domain, password)) {
                authenticationResult.setSuccess();
            }
        }
        catch(Exception e) {
            //TODO
        }
        return authenticationResult;
    }

    @Override
    public IAuthenticationResult isGroupMember(String userid, String domain,
            Role role) {
        IAuthenticationResult authenticationResult = new AuthenticationResult();
        authenticationResult.setFailure();
        try {
            if(SiteSecurity.isAuthenticRole(userid, domain, role.toString())) {
                authenticationResult.setSuccess();
            }
        }
        catch(Exception e) {
            //TODO
        }
        return authenticationResult;
    }

}
```

12.1.2 IAuthenticationManager

Shown below is the interface which must be implemented by your authentication manager.

```
package org.apache.uima.ducc.common.authentication;

public interface IAuthenticationManager {

    /**
```

```
     * This method is expected to return AuthenticationManager implementation version
     * information.  It is nominally displayed by the DUCC webserver on the Login/Logout
     * pages.
     *
     * Example return value: Acme Authenticator 1.0
     *
     * @return The version of the AuthenticationManager implementation.
     */
    public String getVersion();

    /**
     * This method is expected to return password checking information.
     * It is nominally employed by the DUCC webserver to enable/disable
     *  password input area on the Login/Logout pages.
     *
     * @return True if the AuthenticationManager implementation checks passwords;
     * false otherwise.
     */
    public boolean isPasswordChecked();

    /**
     * This method is expected to perform authentication.
     * It is nominally employed by the DUCC webserver for submitted Login pages.
     *
     * @param userid
     * @param domain
     * @param password
     * @return True if authentic userid+domain+password; false otherwise.
     */
    public IAuthenticationResult isAuthenticate(String userid, String domain, String password);

    /**
     * This method is expected to perform role validation.
     * It is nominally employed by the DUCC webserver for submitted Login pages.
     *
     * @param userid
     * @param domain
     * @param role
     * @return True if authentic userid+domain+role; false otherwise.
     */
    public IAuthenticationResult isGroupMember(String userid, String domain, Role role);

    /**
     * The supported Roles
     */
    public enum Role {
        User,
        Admin
    }
}
```

12.1.3 IAuthenticationResult

Shown below is the interface which must be returned by the required authentication methods in your authentication manager.

```
package org.apache.uima.ducc.common.authentication;

public interface IAuthenticationResult {
    public void setSuccess();
    public void setFailure();
    public boolean isSuccess();
    public boolean isFailure();
    public void setCode(int code);
    public int getCode();
    public void setReason(String reason);
    public String getReason();
    public void setException(Exception exception);
    public Exception getException();
}
```

12.1.4 Example ANT script to build jar

Shown below is an example ANT script to build a ducc-authenticator.jar file. The resulting jar file should be placed user DUCC's lib directory along with any dependency jars, and defined in ducc.properties file.

```
<project name="uima-ducc-examples" default="build" basedir=".">

    <property name="TGT-LIB"                value="${basedir}/lib" />
    <property name="TGT-DUCC-AUTH-JAR"      value="${TGT-LIB}/ducc-authenticator.jar" />

    <target name="build" depends="clean, jar" />

    <target name="clean">
        <delete file="${TGT-DUCC-AUTH-JAR}" />
    </target>

    <target name="jar">
        <mkdir dir="${TGT-LIB}" />
        <jar destfile="${TGT-DUCC-AUTH-JAR}" basedir="${basedir}/target/classes/org/apache/uima/ducc/exampl
    </target>

</project>
```

12.1.5 Example ducc.properties entries

Shown here is a snippet of the ducc.properties file defining the class to be used for authentication and the administrator created folder *site-security*, which should contain the ducc-authenticator.jar you built plus any jar files upon which it depends.

Note: the *site-security* directory must be located within DUCC's lib directory.

```
# The class that performs authentication (for the WebServer)
ducc.authentication.implementer = org.apache.uima.ducc.example.authentication.module.AuthenticationManager
```

```
# Site specific jars: include all jars in directory site-security
ducc.local.jars = site-security/*
```

12.1.6 Example ducc.administrators

Example contents of ducc.administrators file located within DUCC's resources directory. Only userids listed here can assume the Administrator role when performing operations via the WebServer.

```
jdoe
fred
hal9000
```

12.2 ducc.properties

The primary configuration file is called ducc.properties and always resides in the directory ducc_runtime/resources.

Some of the properties in ducc.properties are intended as the "glue" that brings the various DUCC components together and lets then run as a coherent whole. These types of properties should be modified only by developers of DUCC itself. In the description below these properties are classified as "Private".

Some of the properties are tuning parameters: timeouts, heartbeat intervals, and so on. These may be modified by DUCC administrators, but only after experience is gained with DUCC, and only to solve specific performance problems. The default tuning parameters have been chosen by the DUCC system developers to provide "best" operation under most reasonable situations. In the description below these properties are classified as "Tuning".

Some of the properties describe the local cluster configuration: the location of the ActiveMQ broker, the location of the Java JRE, port numbers, etc. These should be modified by the DUCC administrators to configure DUCC to each individual installation. In the description below these properties are classified as "Local".

12.2.1 General DUCC Properties

ducc.authentication.implementer

This specifies the class used for WebServer session authentication. If unconfigured, the Web Server enforces no authentication.

Default org.apache.uima.ducc.common.authentication.LinuxAuthenticationManager

Type Local

ducc.admin.endpoint

This is the JMS endpoint name used for DUCC administration messages.

Default ducc.admin.channel

Type Private

ducc.admin.endpoint.type

This is the JMS message type used for DUCC administration requests. If changed DUCC admin may not work.

Default topic

Type Private

ducc.broker.automanage

If set to “true”, DUCC will start and stop the ActiveMQ broker as part of its normal start/stop scripting.

Default true

Type Tuning

ducc.broker.home
For DUCC auto-managed brokers only, this names the location where ActiveMQ is installed installed.

Note that the DUCC installation includes a default ActiveMQ.

Default $DUCC_HOME/activemq

Type Tuning

ducc.broker.memory.options
For DUCC auto-managed brokers only, this names the ActiveMQ configuration file. The configuration file is assumed to reside in the directory specified by *ducc.broker.home*, so the path must be relative to that location.

Default conf/activemq-ducc.xml

Type Tuning

ducc.broker.decoration
The property is used by the DUCC Job Driver processes to modify the ActiveMQ broker URL when connecting to the Job Processes.

The supplied default is used to disable broker connection timeouts. From the ActiveMQ documentation: "The maximum inactivity duration (before which the socket is considered dead) in milliseconds. On some platforms it can take a long time for a socket to appear to die, so we allow the broker to kill connections if they are inactive for a period of time. Use by some transports to enable a keep alive heart beat feature. Set to a value less-than-or-equal0 to disable inactivity monitoring. Declare the wire protocol used to communicate with ActiveMQ."

This decoration is used to keep the broker connection alive while a JVM is in a long garbage collection. The applications that DUCC is designed to support can spend significant time in garbage collection, which can cause spurious timeouts. By default the DUCC configuration disables the timeout by setting it to 0.

Default wireFormat.maxInactivityDuration=0

Type Local

ducc.broker.hostname
This declares the node where the ActiveMQ broker resides. It MUST be updated to the actual node where the broker is running as part of DUCC installation. The default value will not work.

Default ${ducc.head}. The default is defined in the ducc property, *ducc.head*. If you want to run the ActiveMQ broker on the "ducc head", this parameter need not be changed.

Type Local

ducc.broker.jmx.port
This is the port used to make JMX connections to the broker. This should only be changed by administrators familiar with ActiveMQ configuration.

Default 1100

Type Local

ducc.broker.memory.options
For DUCC auto-managed brokers only, this sets the `-Xmx` heap size for the broker.

Default -Xmx2G

Type Tuning

ducc.broker.name
This is the internal name of the broker, used to locate Broker's MBean in JMX Registry. It is NOT related to any node name. When using the ActiveMQ distribution supplied with DUCC it should always be set to "localhost". The default should be changed only by administrators familiar with ActiveMQ configuration.

Default localhost

Type Local

ducc.broker.port
This declares the port on which the ActiveMQ broker is listening for messages. It MAY be updated as part of DUCC installation. ActiveMQ ships with port 61616 as the default port, and DUCC uses that default.

Default 61617

Type Local

ducc.broker.protocol
Declare the wire protocol used to communicate with ActiveMQ.

Default tcp

Type Private

ducc.broker.server.url.decoration
For DUCC auto-managed brokers only, this configures ActiveMQ Server url decoration.

Default transport.soWriteTimeout=45000

Type Tuning

ducc.cli.httpclient.sotimeout
This is the timeout used by the CLI to communicate with DUCC, in milliseconds. If no response is heard within this time, the request times out and is aborted. When set to 0 (the default), the request never times out.

Default 0

Type Tuning

ducc.cluster.name
This is a string used in the Web Server banner to identify the local cluster. It is used for informational purposes only and may be set to anything desired.

Default Apache UIMA-DUCC

Type Local

ducc.head
This property declares the node where the DUCC adminstrative processes run (Orchestrator, Resource Manager, Process Manager, Service Manager). This property is required and MUST be configured in new installation. The installation script ducc_post_install initializes this property to the node the script is executed on.

Default There is no default, this must be configured during system installation.

Type Local

ducc.jms.provider
Declare the type of middleware providing the JMS service used by DUCC.

Default activemq

Type Private

ducc.jmx.port
Every process started by DUCC has JMX enabled by default. When more than one process runs on the same machine this can cause port conflicts. The property ”ducc.jmx.port” is used as the base port for JMX. If the port is busy, it is incremented internally until a free port is found.

The web server's "System − > Daemons" tab is used to find the JMX URL that gets assigned to each of the DUCC management processes. The web server's Job details page for each job is used to find the JMX URL that is assigned to each JP.

Default 2099

Type Private

ducc.jvm

This specifies the full path to the JVM to be used by the DUCC processes. This MUST be configured. The installation script ducc_post_install initializes this property to full path to "java" in the installer's environment. (If the "java" command cannot be found, ducc_post_install exits with error.)

Default None. Must be configured during installation.

Type Local

ducc.node.min.swap.threshold

Specify a minimum amount of free swap space available on a node. If an agent detects free swap space dipping below the value defined below, it will find the fattest (in terms of memory) process in its inventory and kill it. The value of the parameter below is expressed in Megabytes.

If set to 0, the threshold is disabled.

Default 0

Type Tuning

ducc.agent.jvm.args

This specifies the list of arguments passed to the JVM when spawning the Agent.

Default -Xmx100M

Type Tuning

ducc.driver.jvm.args

If enabled, the arguments here are automatically added to the JVM arguments specified for the Job Driver process.

Note: if the user-supplied JVM arguments contain a -Xmx entry then any -Xmx value specified here will be ignored.

Default (unconfigured)

Type Local

ducc.orchestrator.jvm.args

This specifies the list of arguments passed to the JVM when spawning the Orchestrator.

Default -Xmx1G

Type Tuning

ducc.pm.jvm.args

This specifies the list of arguments passed to the JVM when spawning the Process Manager.

Default -Xmx1G

Type Tuning

ducc.process.jvm.args

If enabled, the arguments here are added by DUCC to the JVM arguments in the user's job processes.

Default (unconfigured)

Type Private

ducc.rm.jvm.args

This specifies the list of arguments passed to the JVM when spawning the Resource Manager.

Default -Xmx1G

Type Tuning

ducc.sm.jvm.args

This specifies the list of arguments passed to the JVM when spawning the Service Manager.

Default -Xmx1G

Type Tuning

ducc.ws.jvm.args

specifies the list of arguments passed to the JVM when spawning the Webserver.

Default -Xmx8G

Type Tuning

ducc.locale.language

Establish the language for national language support of messages. Currently only ”en” is supported.

Default en

Type Private

ducc.locale.country

Establish the country for National Language Support of messages. Currently only ”us” is supported.

Default us

Type Private

ducc.runmode

When set to ”Test” this property bypasses userid and authentication checks. It is intended for use ONLY by DUCC developers. It allows developers of DUCC to simulate a multiuser environment without the need for root privileges.

Note: WARNING! Enabling this feature in a production DUCC system is a potentially serious security breach. It should only be set by DUCC developers running with an un-privileged ducc_ling.

Default Unconfigured. When unconfigured, test mode is DISABLED.

Type Local

ducc.signature.required

When set, the CLI signs each request so the Orchestrator can be sure the requestor is actually who he claims to be.

Default on

Type Tuning

ducc.threads.limit

This enforces a maximum number of pipeline threads per job, over all its processes. No job will have more active work-items than this dispatched. This limit is disabled by default.

The value represents the size of the underlying CAS pool in the Job Driver and therefore is related to the size of the Job Driver heap and the real memory consumed by JD. If the JD is consuming too much memory, try setting or reducing this value.

Default (unconfigured)

Type Local

ducc.environment.propagated

This specifies the environmental variables whose values will be merged with the user-specified environment option on job, process and service submissions. A typical setting might be: USER HOME LANG

Default (unconfigured)

Type Local

12.2.2 Web Server Properties

ducc.ws.configuration.class

The name of the pluggable java class used to implement the Web Server.

Default Value org.apache.uima.ducc.ws.config.WebServerConfiguration

Type Private

ducc.ws.node

This is the name of the node the web server is started on. If not specified, the web server is started on `${ducchead}`.

Default Value (unconfigured)

Type Local

ducc.ws.ipaddress

In multi-homed systems it may be necessary to specify to which of the multiple addresses the Web Server listens for requests. This property is an IP address that specifies to which address the Web Server listens.

Default Value (unconfigured)

Type Local

ducc.ws.port

This is the port on which the DUCC Web Server listens for requests.

Default Value 42133

Type Local

ducc.ws.port.ssl

This is the port that the Web Server uses for SSL requests (such as authentication).

Default Value 42155

Type Local

ducc.ws.port.ssl.pw

This is the password used to generate the Web Server's keystore used for HTTPS requests. Usually this keystore is created at initial installation time using ducc_post_install.

Default Value quackquack

Type Local

ducc.ws.session.minutes

Once authenticated, this property determines the lifetime of the authenticated session to the Web Server.

Default Value 60

Type Tuning

ducc.ws.max.history.entries

DUCC maintains a history of all jobs. The state of jobs, both old and current are shown in the Webserver's Jobs Page. To avoid overloading this page and the Web Server, the maximum number of entries that can be shown is regulated by this parameter.

Default Value 4096

Type Tuning

ducc.ws.login.enabled
If true, users are allowed to login to Webserver. If false, users are not allowed to login to Webserver. Shipped value set to false. However, default value if property not specified is true.

Default Value true

Type Tuning

ducc.ws.automatic.cancel.minutes Optionally configure the webserver job automatic cancel timeout. To disable this feature specify 0. This is employed when a user specifies *−−wait_for_completion* flag on job submission, in which case the job monitor program must visit

```
http://<host>:<port>/ducc-servlet/proxy-job-status?id=<job-id>
```

within this expiry time. Otherwise the job will be automatically canceled.

This provides a safeguard against runaway jobs or managed reservations, if the submitter gets disconnected from DUCC in some way.

If the feature is disabled by specifing "0", no work is canceled even if the monitor itself disappears.

Default Value 10

Type Tuning

ducc.ws.jsp.compilation.directory
This specifies the temporary used by the Web Server's JSP engine to compile its JSPs.

Default Value /tmp/ducc/jsp

Type Tuning

12.2.3 Job Driver Properties

ducc.jd.configuration.class
The name of the pluggable java class used to implement the Job Driver.

Default Value org.apache.uima.ducc.jd.config.JobDriverConfiguration

Type Private

ducc.jd.state.update.endpoint
This is the JMS endpoint name by the Job Driver to send state to the Orchestrator.

Default Value ducc.jd.state

Type Private

ducc.jd.state.update.endpoint.type
This is the JMS message type used to send state to the Orchestrator.

Default Value topic

Type Private

ducc.jd.state.publish.rate
The interval in milliseconds between JD state publications to the Orchestrator. A higher rate (smaller number) may slightly increase system response but will increase network load. A lower rate will somewhat decrease system response and lower network load.

Default Value 15000

Type Tuning

ducc.jd.queue.prefix

This is a human-readable string used to form queue names for the JMS queues used to pass CASs from the Job Driver to the Job Processes. The completed queue named comprises the prefix concatenated with the DUCC assigned Job number.

Default Value ducc.jd.queue.

Type Private

ducc.jd.host.class

This is the scheduling class used to request a reservation from the Resource Manager for the machine that will be used to run the Job Driver processes. This class must also be configured in ducc.classes with scheduling policy RESERVE.

Default Value JobDriver

Type Tuning

ducc.jd.host.description

This is a name to be associated with the reservation that is made for the Job Driver Node. It can be any string and is displayed in the Reservations page on the Web Server.

Default Value Job Driver

Type Tuning

ducc.jd.host.memory.size

This is the amount of memory that is requested in the Job Driver reservation. It is used in conjunction with the configuration of the class specified for the job driver (by default, JobDriver) to schedule a node. The default configuration for this class uses a node pool instead of memory to allocate the Job Driver node so by default, this parameter is ignored.

Default Value 2GB

Type Tuning

ducc.jd.host.number.of.machines

This is the number of machines to request for Job Driver nodes. This may be increased if there are many jobs in the system and the load on the JD node is high enough to slow the JD processes.

Default Value 1

Type Tuning

ducc.jd.host.user

This is the userid that is associated with the Job Driver reservation. It does not need to be a ”real” userid as the actual owner of the reservation is user ”ducc”. It is primarily used as annotation of the reservation in the Web Server and logs.

Default Value System

Type Tuning

ducc.jd.queue.timeout.minutes

After dispatching a work item to UIMA-AS client for processing, the number of minutes that the Job Driver will wait for two callbacks (queued and assigned) before considering the work item lost. The elapsed time for the callbacks is normally sub-second. Intermittent network problems may cause unusual spikes.

Default Value 5

Type Tuning

ducc.jd.share.quantum

When CGroups are enabled, this is the RSS, in MB, that is reserved for each JD process, and enforced by the CGroup support. Larger JDs are permitted, but the CGroup support will force the excess RSS onto swap. This potentially slows the performance of that JD, but preserves the resources for other, better-behaved, JDs.

Default Value 400

Type Tuning

12.2.4 Service Manager Properties

ducc.sm.configuration.class
This is the name of the pluggable java class used to implement the Service Manager.

Default Value org.apache.uima.ducc.sm.config.JobDriverConfiguration

Type Private

ducc.sm.state.update.endpoint
This is the JMS endpoint name used for state messages sent by the Service Manager.

Default Value ducc.sm.state

Type Private

ducc.sm.state.update.endpoint.type
This is the JMS message type used for state messages sent by the Service Manager.

Default Value topic

Type Private

ducc.sm.meta.ping.rate
This is the time, in milliseconds, between pings by the Service Manager to each known, running service.

Default Value 60000

Typ Tuning

ducc.sm.meta.ping.stability
This is the number of consecutive pings that may be missed before a service is considered unavailable.

Default Value 10

Type Tuning

ducc.sm.meta.ping.timeout
This is the time in milliseconds the SM waits for a response to a ping. If the service does not respond within this time the ping is accounted for as a ”missed” ping.

Default Value 5000

Type Tuning

ducc.sm.http.port
This is the HTTP port used by the Service Manager to field requests from the CLI / API.

Default Value 19989

Type Local

ducc.sm.http.node
This is the node where the Service Manager runs. It MUST be configured as part of DUCC setup. The *ducc_post_install* procedures initialize this to *${ducc.head}*.

Default Value ${ducc.head}

Type Local

ducc.sm.default.linger

This is the length of time, in milliseconds, that the SM allows a service to remain alive after all jobs that reference it have exited. If no new job referencing it enters the system before this time has expired, the SM stops the service.

Default Value 300000

Type Tuning

ducc.sm.instance.failure.max

This is the maximum number of consecutive failures of service instance initialization permitted before DUCC stops creating new instances. When this cap is hit the SM will disable autostart for the service. It may be overridden by the service registration's *instance_failures_limit* parameter.

Default Value 2

Type Tuning

12.2.5 Orchestrator Properties

ducc.orchestrator. configuration.class

This is the name of the pluggable java class used to implement the DUCC Orchestrator.

Default Value org.apache.uima.ducc.orchestrator.config.OrchestratorConfiguration

Type Private

ducc.orchestrator.start.type

This indicates the level of recovery to be taken on restarting a system. There are two levels of startup:

cold All reservations are canceled, all currently running jobs (if any) are terminated. All services are terminated. The system starts with no jobs, reservations, or services active.

warm All unmanaged reservations are restored. All currently running jobs (if any) are terminated. All services are started or restarted as indicated by their state when the system went down. The system starts with no jobs active, but unmanaged reservations and services are preserved.

Default Value warm

Type Tuning

ducc.orchestrator.state.endpoint

This is the name of the JMS endpoint through which the Orchestrator broadcasts its full state messages. These messages include full job information and can be large. This state is used by the Process Manager and the Webserver.

Default Value ducc.orchestrator.request?requestTimeout=180000

Type Private

ducc.orchestrator.state.update.endpoint.type

This is the JMS endpoint type used for the "full" state messages sent by the Orchestrator.

Default Value topic

Type Private

ducc.orchestrator.state.publish.rate

The interval in milliseconds between Orchestrator publications of its non-abbreviated state.

Default Value 15000

Type Private

ducc.orchestrator.abbreviated.state.endpoint
This is the name of the JMS endpoint through which the Orchestrator broadcasts its abbreviated state. This state state is used by the Resource Manager and Service Manager.

Default Value ducc.orchestrator.abbreviated.state

Type Private

ducc.orchestrator.abbreviated.state.update.endpoint.type
This is the JMS endpoint type used for the "abbreviated" state messages sent by the Orchestrator.

Default Value topic

Type Private

ducc.orchestrator.abbreviated.state.publish.rate
The interval in milliseconds between Orchestrator publications of its abbreviated state.

Default Value 15000

Type Private

ducc.orchestrator.maintenance.rate
This is the interval in milliseconds between Orchestrator maintenance cycles, which check and update history and state.

Default Value 60000

Type Tuning

ducc.orchestrator.http.port
This is the HTTP port used by the Orchestrator to field requests from the CLI / API.

Default Value 19988

Type Local

ducc.orchestrator.http.node
This is the node where the Orchestrator runs. It MUST be configured as part of DUCC setup. The *ducc_post_install* procedures initialize this to *${ducc.head}*.

Default Value ${ducc.head}

Type Local

ducc.orchestrator.unmanaged.reservations.accepted
This flag controls whether the Orchestrator will accept requests for unmanaged reservations (true) or deny request for unmanaged reservations (false).

Default Value true

Type Local

12.2.6 Resource Manager Properties

ducc.rm.configuration.class
This is the name of the pluggable java class used to implement the DUCC Resource Manager.

Default Value org.apache.uima.ducc.rm.config.ResourceManagerConfiguration

Type Private

ducc.rm.state.update.endpoint
This is the name of the JMS endpoint through which the Resource Manager broadcasts its abbreviated state.

Default Value ducc.rm.state

Type Private

ducc.rm.state.update.endpoint.type

This is the JMS endpoint type used for state messages sent by the Resource Manager.

Default Value topic

Type Private

ducc.rm.state.publish.rate

The interval in milliseconds between Resource Manager state publications to the Orchestrator.

This can directly affect user response. For small clusters a value of 10000 is often acceptable and results in faster response. On larger clusters the value should be raised to around 60000 (60 seconds) to avoid thrashing.

Default Value 10000

Type Tuning

ducc.rm.share.quantum

The share quantum is the smallest amount of RAM that is schedulable for jobs, in GB. Jobs are scheduled based entirely on their memory requirements. Memory is allocated in multiples of the share quantum.

See the Resource Management section for more information on the share quantum.

Default Value 1

Type Tuning

ducc.rm.scheduler

The component that implements the scheduling algorithm is pluggable. This specifies the name of that class.

Default Value org.apache.uima.ducc.rm.scheduler.NodepoolScheduler

Type Private

ducc.rm.class.definitions

This specifies the name of the file that contains the site's class definitions. This file is described in detail the section on ducc.properties [77].

Default Value ducc.classes

Type Tuning

ducc.rm.node.stability

The RM receives regular "heartbeats" from the DUCC agents in order to know what nodes are available for scheduling. The node.stability property configures the number of consecutive heartbeats that may be missed before the Resource Manager considers the node to be inoperative.

If a node becomes inoperative, the Resource Manager deallocates all processes on that node and attempts to reallocate them on other nodes. The node is marked offline and is unusable until its heartbeats start up again.

The default configuration declares the agent heartbeats to occur at 1 minute intervals. Therefore heartbeats must be missed for five minutes before the Resource Manager takes corrective action.

Default Value 5

Type Tuning

ducc.rm.init.stability

During DUCC initialization the Resource Manager must wait some period of time for all the nodes in the cluster to check-in via their "heartbeats". If the RM were to start scheduling too soon there would be a period of significant "churn" as the perceived cluster configurations changes rapidly. As well, it would be impossible to recover work in a warm or hot start if the affected nodes had not yet checked in.

The init.stability property indicates how many heartbeat intervals the RM must wait before it starts scheduling after initialization.

Default Value 2

Type Tuning

ducc.rm.eviction.policy

The alternative value is SHRINK_BY_MACHINE.

The eviction.policy is a heuristic to choose which processes of a job to preempt because of competition from other jobs.

The SHRINK_BY_INVESTMENT policy attempts to preempt processes such that the

least amount of work is lost. It chooses candidates for eviction in order of:

1. Processes still initializing, with the smallest time spent in the initializing step.
2. Processes whose currently active work items have been executing for the shortest time.

The SHRINK_BY_MACHINE policy attempts to preempt processes so as to minimize fragmentation on machines with large memories that can contain multiple job processes. No consideration of execution time or initialization time is made.

Default Value SHRINK_BY_INVESTMENT

Type Tuning

ducc.rm.initialization.cap

The type of jobs supported by DUCC generally have very long and often fragile initialization periods. Errors in the applications and other problems such is missing or errant services can cause processes to fail during this phase.

To avoid preempting running jobs and allocating a large number of resources to jobs only to fail during initialization, the Resource Manager schedules a small number of processes until it is determined that the initialization phase will succeed.

The initialization.cap determines the maximum number of processes allocated to a job until at least one process successfully initializes. Once any process initializes the Resource Manager will proceed to allocate the job its full fair share of processes.

The initialization cap can be overridden on a class basis by configuration via ducc.classes.

Default Value 1

Type Tuning

ducc.rm.expand.by.doubling

When a job expands because its fair share has increased, or it has completed initialization, it may be desired to govern the rate of expansion. If expand.by.doubling is set to ”true”, rather than allocate the full fair share of processes, the number of processes is doubled each scheduling cycle, up to the maximum allowed.

Expand.by.doubling can be overridden on a class basis by configuration via ducc.classes.

Default Value true

Type Tuning

ducc.rm.prediction

Because initialization time may be very long, it may be the case that a job that might be eligible for expansion will be able to complete in the currently assigned shares before any new processes are able to complete their initialization. In this case expansion results in waste of resources and potential eviction of processes that need not be evicted.

The Resource Manager monitors the rate of task completion and attempts to predict the maximum number of processes that will be needed at a time in the future based on the known process initialization time. If it is determined that expansion is unnecessary then it is not done for the job.

Prediction can be overridden on a class basis by configuration via ducc.classes.

Default Value true

Type Tuning

ducc.rm.prediction.fudge

When ducc.rm.prediction is enabled, the known initialization time of a job's processes plus some "fudge" factor is used to predict the number of future resources needed. The "fudge" is specified in milliseconds.

The default "fudge" is very conservative. Experience and site policy should be used to set a more practical number.

Prediction.fudge can be overridden on a class basis by configuration via ducc.classes.

Default Value 120000

Type Tuning

ducc.rm.defragmentation.threshold

If *ducc.rm.defragmentation* is enable, limited defragmentation of resources is performed by the Resource Manager to create sufficient space to schedule work that has insufficient resources (new jobs, for example.). The term *insufficient* is defined as "needing more processes than the defragmentation threshold, but currently having fewer processes than the defragmentation threshold." These are called "needy" jobs. Additionally, the Resource Manager will never evict processes from "needy" jobs for the purpose of defragmentation.

This property allows installations to customize the value used to determine if a job is "needy". Jobs with fewer processes than this are potentially needed, and jobs with more processes are never needy.

Default Value 2

Type Tuning

12.2.7 Agent Properties

ducc.agent.configuration.class

This is the name of the pluggable java class used to implement the DUCC Agents.

Default Value org.apache.uima.ducc.nodeagent.config.AgentConfiguration

Type Private

ducc.agent.request.endpoint

This is the JMS endpoint through which agents receive state from the Process Manager.

Default Value ducc.agent

Type Private

ducc.agent.request.endpoint.type

This is the JMS endpoint type used for state messages sent by the Process Manager.

Default Value topic

Type Private

ducc.agent.managed.process.state.update.endpoint

This is the JMS endpoint used to communicate from the managed process to the Agent (Job Process).

Default Value ducc.managed.process. state.update

Type Private

ducc.agent.managed.process.state.update.endpoint.type

This is the JMS endpoint type used to communicate from the managed process (Job Process) to the Agent.

Default Value socket

Type Private

ducc.agent.managed.process. state.update.endpoint.params

These are configuration parameters for the Agent-to-JP communication socket. These should only be modified by DUCC developers.

Default Value transferExchange=true&sync=false

Type Private

ducc.agent.node.metrics.endpoint

This is the JMS endpoint used to send node metrics updates to listeners. Listeners are usually the Resource Manager and Web Server. These messages serve as node ”heartbeats”. As well, the node metrics heartbeats contain the amount of RAM on the node and the number of processors.

Default Value ducc.node.metrics

Type Private

ducc.agent.node.metrics.endpoint.type

This is the JMS endpoint type used to send node metrics updates from the agents.

Default Value topic

Type Private

ducc.agent.node.metrics.publish.rate

The interval in milliseconds between node metric publications. Every agent publishes its updates at this rate. On large clusters, a high rate (small interval) can be a burden on the network.

Note: the Resource Manager uses the data in the node metrics for scheduling.

Default Value 60000

Type Tuning

ducc.agent.node.inventory.endpoint

This is the JMS endpoint used to send node inventory messages to listeners. Listeners are usually the Orchestrator and Web Server. Information in these messages include a map of processes being managed on the node.

Default Value ducc.node.inventory

Type Private

ducc.agent.node.inventory.endpoint.type

This is the JMS endpoint type used to send node inventory updates from the agents.

Default Value topic

Type Private

ducc.agent.node.inventory.publish.rate

The interval in milliseconds between node inventory publications.

If the inventory has not changed since the last update the agent bypasses sending the update, up to a maximum of ducc.agent.node.inventory.publish.rate.skip times.

Default Value 10000

Type Tuning

ducc.agent.node.inventory.publish.rate.skip

This is the number of times the agent will bypass publishing its node inventory if the inventory has not changed.

Default Value 30

Type Tuning

ducc.agent.launcher.thread.pool.size

This is establishes the size of the agent's threadpool used to manage spawned processes.

Default Value 10

Type Tuning

ducc.agent.launcher.use.ducc.spawn

This specifies whether to launch job processes via ducc_ling. When set to false the process is launched directly as a child of the agent. Log indirection is not performed, the working directory is not set, and the the process does not change its identity to that of the submitter. This property is intended for the use of DUCC developers.

Default Value true

Type Private

ducc.agent.launcher.ducc_spawn_path

This property specifies the full path to the ducc_ling utility. During installation ducc_ling is normally moved to local disk and given setuid-root privileges. Use this property to tell the DUCC agents the location of the installed ducc_ling.

Default Value $DUCC_HOME/admin/ducc_ling

Type Tuning

ducc.agent.launcher.process.stop.timeout

This property specifies the time, in milliseconds, the agent should wait before forcibly terminating a job process (JP) after an attempted graceful shutdown. If the child process does not terminate in the specified time, it is forcibly terminated with kill -9.

This type of stop can occur because of preemption or system shutdown.

Default Value 60000

Type Tuning

ducc.agent.launcher.process.init.timeout

This property specifies the time, in milliseconds, that the agent should wait for a job process (JP) to complete initialization. If initialization is not completed in this time the process is terminated and and Initialization-Timout status is send to the job driver (JD) which decides whether to retry the process or terminate the job.

Default Value 7200000

Type Tuning

ducc.agent.share.size.fudge.factor

The DUCC agent monitors the size of the resident memory of its spawned processes. If a process exceeds its declared memory size by any significant amount it is terminated and a ShareSizeExceeded message is sent. The Job Driver counts this towards the maximum errors for the job and will eventually terminate the job if excessive such errors occur.

This property defines the percentage over the declared memory size that a process is allowed to grow to before being terminated.

To disable this feature, set the value to -1.

Default Value 5

Type Tuning

ducc.agent.rogue.process.user.exclusion.filter

The DUCC Agents scan nodes for processes that should not be running; for example, a job may have left a 'rogue' process alive when it exits, or a user may log in to a node unexpectedly. These processes are reported to the administrators via the webserver for possible action.

This configuration parameter enumerates userids which are ignored by the rogue-process scan.

Default Value root,posstfix,ntp,nobody,daemon,100

Type Tuning

ducc.agent.rogue.process.exclusion.filter

The DUCC Agents scan nodes for processes that should not be running; for example, a job may have left a 'rogue' process alive when it exits, or a user may log in to a node unexpectedly. These processes are reported to the administrators via the webserver for possible action.

This configuration parameter enumerates processes by name which are ignored by the rogue process detector.

Default Value sshd:,-bash,-sh,/bin/sh,/bin/bash,grep,ps

Type Tuning

ducc.agent.launcher.cgroups.enable

Enable or disable CGroups support. If CGroups are not installed on a specific machine, this is ignored.

With CGroups the RSS for a managed process (plus any children processes it may spawn) is limited to the allocated share size. Additional memory use goes to swap space. DUCC monitors and limits swap use to the same proportion of total swap space as allocated share size is to total RAM. If a process exceeds its allowed swap space it is terminated and a ShareSizeExceeded message is sent to the Job Driver.

Nodes not using CGroups fall back to the ducc.agent.share.size.fudge.factor.

Default Value true

Type Tuning

ducc.agent.launcher.cgroups.utils.dir

Location of CGroups programs, like cgexec. If CGroups are not installed on a specific machine, this is ignored.

Depending on the OS, CGroups programs may be installed in different places. Provide a comma separated list of directories the agent should search to find the programs.

Default Value /usr/bin

Type Tuning

ducc.agent.exclusion.file

This specifies the exclusion file to enable node based exclusion for various features. Currently only CGroup exclusion is supported.

The exclusion file has one line per agent of the form:

```
<node>=cgroups
```

If the keyword “cgrouops” is found, the node is excluded from CGroup support.

Default Value Not configured.

Type Tuning

12.2.8 Process Manager Properties

ducc.pm.configuration.class

This is the name of the pluggable java class used to implement the DUCC Process Manager.

Default Value org.apache.uima.ducc.pm.config.ProcessManagerConfiguration

Type Private

ducc.pm.request.endpoint

This is the endpoint through which process manager receive state from the Orchestrator.

Default Value ducc.pm

Type Private

ducc.pm.request.endpoint.type

This is the JMS endpoint type used for state messages sent by the Orchestrator.

Default Value queue

Type Private

ducc.pm.state.update.endpoint

This is the endpoint through which process manager sends its heartbeat. The main receiver is the Web Server for it's daemon status page.

Default Value ducc.pm.state

Type Private

ducc.pm.state.update.endpoint.type

This is the JMS endpoint type used for process manager heartbeats. The primary receiver is the Web Server for its daemon status page.

Default Value topic

Type Private

ducc.pm.state.publish.rate

The interval in milliseconds between process manager heartbeat publications.

Default Value 25000

Type Private

12.2.9 Job Process Properties

ducc.uima-as.configuration.class

This is the name of the pluggable java class that implements the the UIMA-AS service shell for job processes (JPs).

Default Value org.apache.uima.ducc.agent.deploy.uima.UimaAsServiceConfiguration

Type Private

ducc.uima-as.endpoint

This is the endpoint through which job processes (JPs) receive messages from the Agents.

Default Value ducc.job.managed.service

Type Private

ducc.uima-as.endpoint.type

This is the JMS endpoint type used for messages sent to the JPs from the Agents.

Default Value socket

Type Private

ducc.uima-as.endpoint.params

This configures the JP-to-Agent communication socket. It should be changed only by DUCC developers.

Default Value transferExchange=true&sync=false

Type Private

ducc.uima-as.saxon.jar.path

This configures the path the required Saxon jar.

Default Value file:$DUCC_HOME/lib/saxon/saxon8.jar

Type Private

ducc.uima-as.dd2spring.xsl.path

This configures the path the required dd2spring xsl definitions.

Default Value $DUCC_HOME/resources/dd2sprint.xsl

Type Private

ducc.uima-as.flow_controller.specifier

This configures the pluggable class that implements the default flow controller used in the DUCC job processes (JPs).

Default Value DuccJobProcessFC

Type Private

12.3 Resource Manager Configuration: Classes and Nodepools

The class configuration file is used by the Resource Manager configures the rules used for job scheduling. See the Resource Manager chapter for a detailed description of the DUCC scheduler, scheduling classes, and how classes are used to configure the scheduling process.

The scheduler configuration file is specified in ducc.properties. The default name is ducc.classes and is specified by the property *ducc.rm.class.definitions*.

12.3.1 Nodepools

Overview

A *nodepool* is a grouping of a subset of the physical nodes to allow differing scheduling policies to be applied to different nodes in the system. Some typical nodepool groupings might include:

1. Group Intel and Power nodes separately so that users may submit jobs that run only in Intel architecture, or only Power, or “don’t care”.
2. Designate a group of nodes with large locally attached disks such that users can run jobs that require those disks.
3. Designate a specific set of nodes with specialized hardware such as high-speed network, such that jobs can be scheduled to run only on those nodes.

A Nodepool is a subset of some larger collection of nodes. Nodepools themselves may be further subdivided. Nodepools may not overlap: every node belongs to exactly one nodepool. During system start-up the consistency of nodepool definition is checked and the system will refuse to start if the configuration is incorrect.

For example, the diagram below is an abstract representation of all the nodes in a system. There are five nodepools defined:

- Nodepool “NpAllOfThem” is subdivided into three pools, NP1, NP2, and NP3. All the nodes not contained in NP1, NP2, and NP3 belong to the pool called “NpAllOfThem”.
- Nodepool NP1 is not further subdivided.
- Nodepool NP2 is not further subdivided.
- Nodepool NP3 is further subdivided to form NP4. All nodes within NP3 but not in NP4 are contained in NP3.

- Nodepool NP4 is not further subdivided.

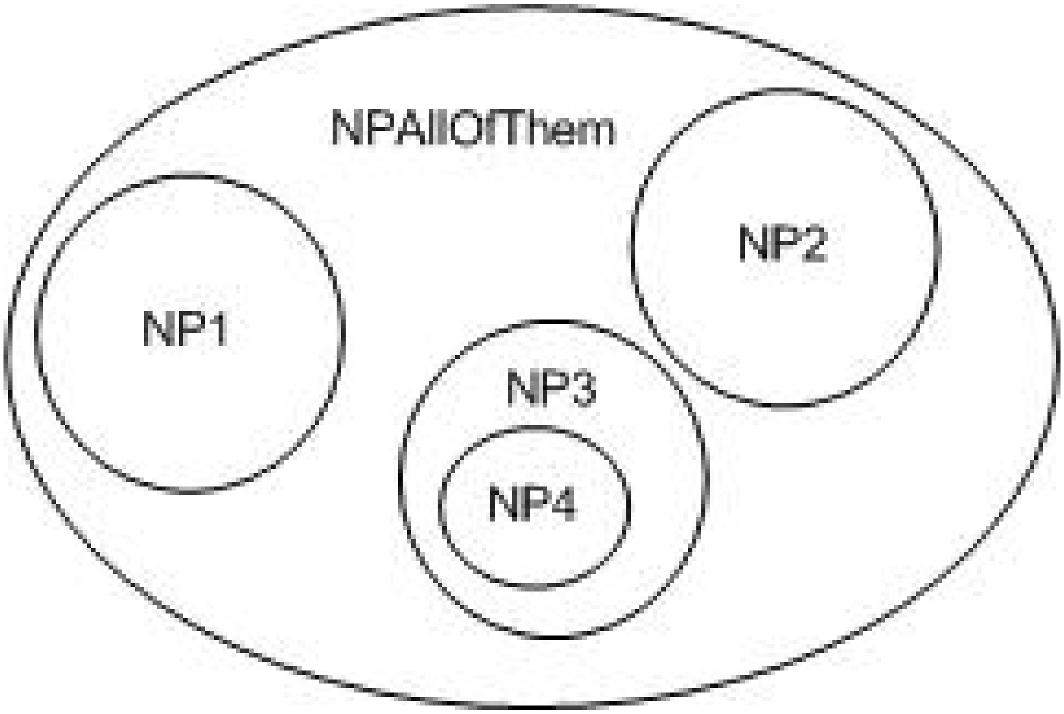


Figure 12.1: Nodepool Example

In the figure below the Nodepools are incorrectly defined for two reasons:

1. NP1 and NP2 overlap.
2. NP4 overlaps both nodepool “NpAllOfThem” and NP3.

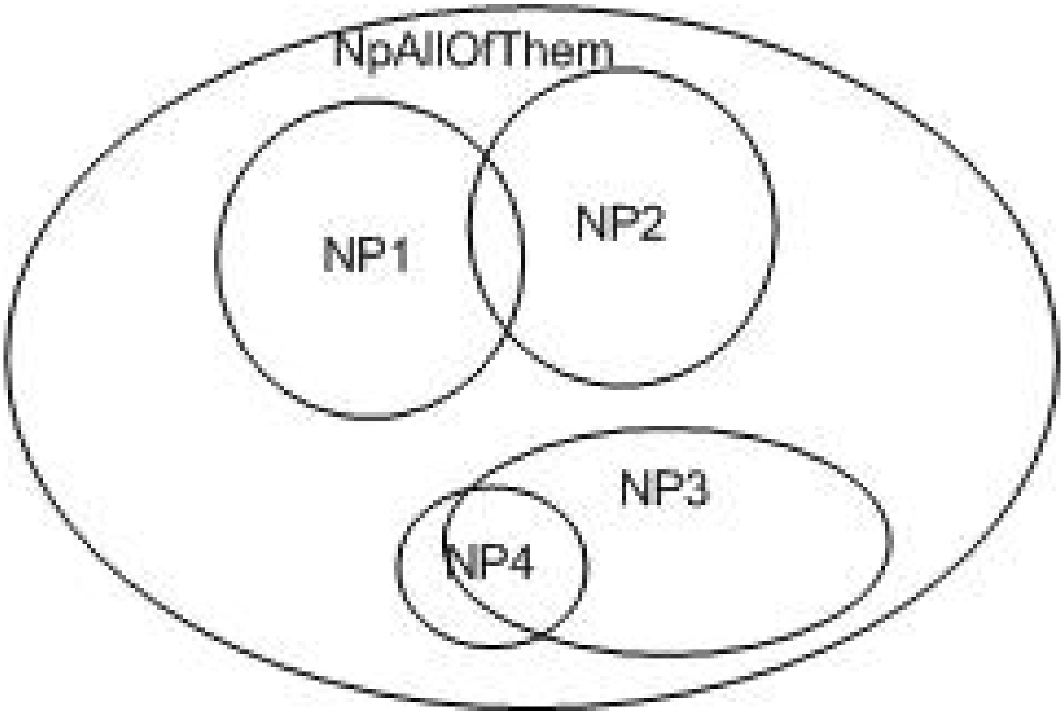


Figure 12.2: Nodepools: Overlapping Pools are Incorrect

Multiple "top-level" nodepools are allowed. A "top-level" nodepool has no containing pool. Multiple top-level pools logically divide a cluster of machines into *multiple independent clusters* from the standpoint of the scheduler. Work scheduled over one pool in no way affects work scheduled over the other pool. The figure below shows an abstract nodepool configuration with two top-level nodepools, "Top-NP1" and "Top-NP2".

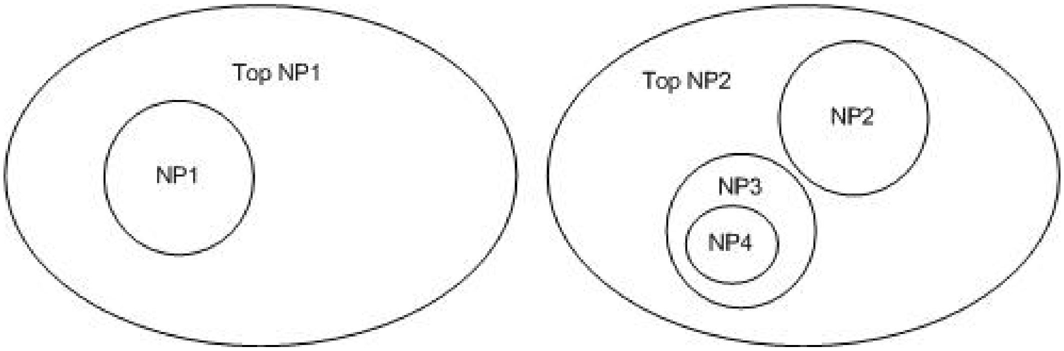


Figure 12.3: Nodepools: Multiple top-level Nodepools

Scheduling considerations

A primary goal of the scheduler is to insure that no resources are left idle if there is pending work that is able to use those resources. Therefore, work scheduled to a class defined over a specific nodepool (say, NpAllOfThem), may be scheduled on nodes in any of the nodepools contained within NpAllOfThem. If work defined over a subpool (such as NP1) arrives, processes on nodes in NP1 that were scheduled for NpAllOfThem are considered *squatters* and are the most likely candidates for eviction. (Processes assigned to their proper nodepools are considered *residents* and are evicted only after all *squatters* have been evicted.) The scheduler strives to avoid creating *squatters*.

Because non-preemptable allocations can't be preempted, work submitted to a class implementing one of the non-preemptable policies (FIXED or RESERVE) are never allowed to "squat" in other nodepools and are only scheduled on nodes in their proper nodepool.

In the case of multiple top-level nodepools: these nodepools and their sub-pools form independent scheduling groups. Specifically,

- Fair-share allocations over any nodepool in one top-level pool do NOT affect the fair-share allocations for jobs in any other top-level nodepool.
- Work submitted to classes under one top-level nodepool do NOT get expanded to nodes under another top-level nodepool, even is there is sufficient capacity.

Most installations will want to assign the majority of nodes to a single top-level nodepool (or its subpools), using other top-level pools for nodes that cannot be shared with other work.

Configuration

DUCC uses simple named stanzas containing key/value pairs to configure nodepools.

At least one nodepool definition is required. This nodepool need not have any subpools or node definitions. The first top-level nodepool is considered the "default" nodepool. Any node not named specifically in one of the node files which checks in with DUCC is assigned to this first, *default* nodepool.

Thus, if only one nodepool is defined with no other attributes, all nodes are assigned to that pool.

A nodepool definition consists of the token "Nodepool" followed by the name of the nodepool, followed by a block delimited with "curly" braces { and }. This block contains the attributes of the nodepool as key/value pairs. Lineneds are ignored. A semicolon ";" may optionally be used to delimit key/value pairs for readability, and an equals sign "=" may optionally be used to delimit keys from values, also just for readability. See the below.

The attributes of a Nodepool are:

domain This is valid only in the "default" (first) nodepool. Any node in any nodefile which does not have a domain, and any node which checks in to the Resource Manager without a domain name is assigned this domain name in order that the scheduler may deal entirely with full-qualified node names.

If no *domain* is specified, DUCC will attempt to guess the domain based on the domain name returned on the node where the Resource Manager resides.

nodefile This is the name of a file containing the names of the nodes which are members of this nodepool.

parent This is used to indicate which nodepool is the logical parent. Any nodepool without a *parent* is considered a top-level nodepool.

The following example defines six nodepools,

1. A top-level nodepool called "–default–". All nodes not named in any nodefile are assigned to this nodepool.
2. A top-level nodepool called "jobdriver", consisting of the nodes named in the file *jobdriver.nodes*.
3. A subpool of "–default–" called "intel", consisting of the nodes named in *intel.nodes*.
4. A subpool of "–default–" called "power", consisting of the nodes named in the file *power.nodes*.
5. A subpool of "intel" called "nightly-test", consisting of the nodes named in *nightly-test.nodes*.
6. And a subpool of "power" called "timing-p7", consisting of the nodes named in *timing-p7.nodes*.

```
Nodepool --default--  { domain mydomain.net }
Nodepool jobdriver    { nodefile jobdriver.nodes }

Nodepool intel        { nodefile intel.nodes        ; parent --default-- }
Nodepool power        { nodefile power.nodes        ; parent --default-- }

Nodepool nightly-test { nodefile nightly-test.nodes ; parent intel }
Nodepool timing-p7    { nodefile timing-p7.nodes    ; parent power }
```

Figure 12.4: Sample Nodepool Configuration

12.3.2 Class Definitions

Scheduler classes are defined in the same simple block language as nodepools.

A simple inheritance (or "template") scheme is supported for classes. Any class may be configured to "derive" from any other class. In this case, the child class acquires all the attributes of the parent class, any of which may be selectively overridden. Multiple inheritance is not supported but nested inheritance is; that is, class A may inherit from class B which inherits from class C and so on. In this way, generalized templates for the site's class structure may be defined.

The general form of a class definition consists of the keyword Class, followed by the name of the class, and then optionally by the name of a "parent" class whose characteristics it inherits. Following the name (and optionally parent class name) are the attributes of the class, also within a { block } as for nodepools, and with lines and key/value pairs optionally delimited by ";" and "=", respectively. See the sample below.

The attributes defined for classes are:

abstract If specified, this indicates this class is a template ONLY. It is used as a model for other classes. Values are "true" or "false". The default is "false". This class is never passed to the scheduler and may not be referenced by jobs.

cap This specifies the largest number of shares that any job in this class may be assigned. It may be an absolute number or a percentage. If specified as a percentage (i.e. it contains a trailing %), it specifies a percentage of the total nodes in the containing nodepool.

debug FAIR_SHARE only. This specifies the name of a class to substitute for jobs submitted for debug. For example, if class *normal* specifies

```
debug = fixed
```

then any job submitted to this class with debugging requested is actually scheduled in class *fixed.* (For example, one probably does not want a debugging job scheduled as FAIR_SHARE and possibly preempted, preferring the non-preemptable class *fixed.*

default This specifies the class to be used as the default class for work submission if no class is explicitly given. Only one class of type FAIR_SHARE may contain this designation, in which case it names the default FAIR_SHARE class. Only one class of type FIXED_SHARE or RESERVE may contain this designation, in which case it names the default class to use for reservations (Note that either FIXED_SHARE or RESERVE scheduling policies are valid for reservations.)

enforce RESERVE only. If specified, then reservations for this class must specify a memory size that when converted to shares exactly matches an eligible machine. For example, if the share quantum is 15GB, a 15GB reservation will never be honored by a 240GB machine; for this machine a memory size between 226 & 240GB would have to be specified. The DUCC Web Server's *Machines* page displays the recommended request size for every machine.

If *enforce* is not specified, the default is "true".

If *enforce* is set to false, the scheduler will attempt to match the reservation as closely as possible to an existing machine, and if it cannot it will use the next largest machine available. Thus, a 15GB reservation *might* be satisfied with a 240GB machine if that is all that is available at the time.

expand-by-doubling FAIR_SHARE only. If "true", and the *initialization-cap* is set, then after any process has initialized, the job will expand to its maximum allowable shares by doubling in size each scheduling cycle.

If not specified, the global value set in ducc.properties is used.

initialization-cap FAIR_SHARE only. If specified, this is the largest number of processes this job may be assigned until at least one process has successfully completed initialization.

If not specified, the global value set in ducc.properties is used.

max-processes FAIR_SHARE and FIXED_SHARE only. This is the largest number of FIXED-SHARE, non-preemptable shares any single job may be assigned.

Omit this property, or set it to 0 to disable the cap.

max-machines RESERVE only. This specifies the maximum number of full machines that may be reserved by any single job.

Omit this property, or set it to 0 to disable the cap.

prediction-fudge FAIR_SHARE only. When the scheduler is considering expanding the number of processes for a job it tries to determine if the job may complete before those processes are allocated and initialized. The *prediction-fudge* adds some amount of time (in milliseconds) to the projected completion time. This allows installations to prevent jobs from expanding when they were otherwise going to end in a few minutes anyway.

If not specified, the global value set in ducc.properties is used.

nodepool If specified, jobs for this class are assigned to nodes in this nodepool. The value must be the name of one of the configured nodepools.

policy This is the scheduling policy, one of FAIR_SHARE, FIXED_SHARE, or RESERVE. This attribute is required (there is no default).

priority This is the scheduling priority for jobs in this class.

weight FAIR_SHARE only. This is the fair-share weight for jobs in this class.

The following figure illustrates a representative class configuration for a large cluster, consisting of mixed Intel and Power nodes. This class definition assumes the nodepool configuration shown above. FAIR_SHARE, FIXED_SHARE, and RESERVE classes are defined over each machine architecture, Intel and Power, and over the combined pool.

```
# --------------------- Fair share definitions ---------------
Class fair-base {
      policy = FAIR_SHARE
      nodepool = intel
      priority = 10
      weight = 100
      abstract = true
      debug = fixed
}

Class nightly-test   fair-base  { weight = 100; nodepool nightly-test; priority = 7}

Class background     fair-base  { weight = 20 }
Class low            fair-base  { weight = 50 }
Class normal         fair-base  { weight = 100; default = true }
Class high           fair-base  { weight = 200 }
Class weekly         fair-base  { weight = 400 }

Class background-p7  background { nodepool = power }
Class low-p7         low        { nodepool = power }
Class normal-p7      normal     { nodepool = power }
Class high-p7        high       { nodepool = power }
Class weekly-p7      weekly     { nodepool = power }

Class background-all background { nodepool = --default-- }
Class low-all        low        { nodepool = --default-- }
Class normal-all     normal     { nodepool = --default-- }
Class high-all       high       { nodepool = --default-- }
Class weekly-all     weekly     { nodepool = --default-- }

# --------------------- Fixed share definitions ---------------
Class fixed-base {
      policy = FIXED_SHARE
      nodepool = intel
      priority = 5
      abstract = true
      max-processes = 10
}

Class fixed     fixed-base { }
Class fixed-p7  fixed-base { nodepool = power;    default = true; }
Class JobDriver fixed-base { nodepool = jobdriver; priority = 0 }

# --------------------- Reserve definitions ---------------
Class reserve-base {
      policy = RESERVE
      nodepool = intel
      priority = 1
      enforce = true
      abstract = true
      max-machines = 10
}

Class reserve     reserve-base { }
Class reserve-p7  reserve-base { nodepool = power }
Class timing-p7   reserve-base { nodepool = timing-p7 }
```

Figure 12.5: Sample Class Configuration

12.3.3 Validation

The administrative command, *check_ducc* may be used to validate a configuration, with the *-c* option. This reads the entire configuration and nodefiles, validates consistency of the definitions and insures the nodepools do not overlap.

During start-up, the *start_ducc* command always runs full validation, and if the configuration is found to be incorrect, the cluster is not started.

Configuration checking is done internally by the DUCC java utility *org.apache.uima.ducc.commonNodeConfiguration*. This utility contains a public API as described in the Javadoc. It may be invoked from the command line as follows:

Usage:

```
java org.apache.uima.ducc.commonNodeConfiguration [-p] [-v nodefile] configfile
```

Options:

−p Pretty-print the compiled configuration to stdout. This illustrates nodepool nesting, and shows the fully-completed scheduling classes after inheritance.

−v **nodefile** This should be the master nodelist used to start DUCC. This is assumed to be constructed to reflect the nodepool organization as described here. If provided, the nodepools are validated and checked for overlaps.

configfile This is the name of the file containing the configuration.

12.4 Ducc Node Definitions

The DUCC node definitions are specified by default in the file *ducc.nodes*.

The DUCC node list is used to configure the nodes used to run jobs and assign reservations. When DUCC is started, the nodelist is read an a DUCC Agent is started on every node in the list.

The node list can be composed of multiple node lists to assist organization of the DUCC cluster. All the administrative commands operate upon node lists. By carefully organized these lists it is possible to administer portions of a cluster independently.

In particular, it is highly recommended that the nodelists reflect the nodepool structure. In this way, the configuration used to start DUCC is guaranteed to match the nodeppool definitions.

Several types of records are permitted in nodelists:

Comments A comment starts with the symbol "#". All text on the line following this symbol is ignored.

import If a line starts with the symbol *import*, the next symbol on that line is expected to be the name of another node list. This permits the DUCC cluster's nodes to be configured in a structured manner.

For instance, the file *ducc.nodes* might consist entirely of *import* statements naming all of the nodepool files.

domain This must be the first line of the file. If specified, it should name the default domain to be used for all the nodes in this file, and the nodes named in imported files. If not specified, then during start-up, nodes without domain names are assigned domain names according to the global domain name specified in the Resource Manager configuration file, and if none is specified there, the domain name on the host starting DUCC is used.

nodename This is a single token consisting of the name of a node on which an agent it to be started.

The example below shows a partial, hypothetical node configuration corresponding to the nodepool configuration above.

```
> cat ducc.nodes
# import all the nodes corresponding to my nodepools
domain my.domain
import intel.nodes
import power.nodes
import jobdriver.nodes
import nightly-test.nodes
import timing-p7.nodes

> cat intel.nodes
# import the intel nodes, by frame
import intel-frame1.nodes
import intel-frame2.nodes
import intel-frame3.nodes

>cat intel-frame1.nodes
#import the specific nodes from frame1
r1s1node1
r1s1node2
r1s1node3
r1s1node4
```

Figure 12.6: Sample Node Configuration

12.5 Administrative Commands

The administrative commands include a command to start DUCC, one to stop it, and one to verify the configuration and query the state of the cluster.

Note: The scripting that supports these functions runs (by default) in multi-threaded mode so large clusters can be started, stopped, and queried quickly. If DUCC is running on an older system, the threading may not work right, in which case the scripts detect this and run single-threaded. As well, all these commands support a "–nothreading" option to manually disable the threading.

12.5.1 start_ducc

Description

The command $DUCC_HOME/admin/start_ducc is used to start DUCC processes. If run with no parameters it takes the following actions:

- Starts the management processes Resource Manager, Orchestrator, Process Manager, Services Manager, and Web Server on the local node (where start_ducc is executed.
- Starts an agent process on every node named in the default node list.

Usage

start_ducc [options]
If no options are given, all DUCC processes are started, using the default node list, *ducc.nodes*.

Options:

-n, –nodelist [**nodefile**]
Start agents on the nodes in the nodefile. Multiple nodefiles may be specified:

```
start\_ducc -n foo.nodes -n bar.nodes -n baz.nodes
```

-s, –singleuser
Start DUCC in "single-user" mode. This inhibits certain checks that are ususally needed running in multi-user mode; specifically, permissions on the ducc_ling utility are not verified. In this mode all jobs are usually run as whatever user is used to start DUCC.

-c, –component [**component**]
Start a specific DUCC component, optionally on a specific node. If the component name is qualified with a nodename, the component is started on that node. To qualify a component name with a destination node, use the notation component@nodename. Multiple components may be specified:

```
start\_ducc -c sm -c pm -c rm -c or@bj22 -c agent@n1 -c agent@n2
```

Components include:

rm The Resource Manager

or The Orchestrator

pm The Process Manager

sm The Service Manager

ws The Web Server

agent Node Agents

–**nothreading** If specified, the command does not run in multi-threaded mode even if it is supported on the local platform.

Notes:

A different nodelist may be used to specify where Agents are started. As well multiple node lists may be specified, in which case Agents are started on all the nodes in the multiple node lists.

To start only agents, run start_ducc specifying a nodelist explicitly. When started like this, the management daemons are not started unless explicitly requested.

To start only management processes, run start_ducc with the -m or –management flags. When started like the the agents are not started unless explicitly requested.

To start a specific management process, run start_ducc with the -c component parameter, specify the component that should be started.

Examples:

Start some nodes from two different nodelist. This doesn't start any of the management processes but it does insure the ActiveMQ Broker is available.

```
start\_ducc -n foo.nodes -n bar.nodes
```

Start an agent on a specific node:

```
start\_ducc -c agent@a.specific.node
```

Start the webserver on node 'bingle':

```
start\_ducc -c ws@bingle
```

Debugging:

Sometimes something will not start and it can be difficult to understand why. To diagnose, it is helpful to know that *start_ducc* is simply a wrapper around a lower-level bit of scripting that does the actual work. That lower-level code can be invoked stand-alone, in which case console messages that *check_ducc* will have suppressed are presented to the console.

The lower-level script is called *ducc.py* and accepts the same *-c component* flag as start_ducc. If some component will not start, try running *ducc.py -c component* directly. It will start in the foreground and usually the cause of the problem becomes evident from the console.

For example, suppose the Resource Manager will not start. Run the following:

```
./ducc.py -c rm
```

and examine the output. Use *CTL-C* to stop the component when done.

12.5.2 stop_ducc

Description:

Stop_ducc is used to stop DUCC processes. If run with no parameters it takes the following actions: TODO: Garbled by maven or docbook, update this

Usage:

stop_ducc [options]
If no options are given, help text is presented. At least one option is required, to avoid accidental cluster shutdown.

Options:

-a –all
Stop all the DUCC processes, including agents and management processes. This broadcasts a "shutdown" command to all DUCC processes. Shutdown is normally performed gracefully will all process including job processes given time to save state. All user processes, both jobs and services, are sent shutdown signals. Job and service processes which do not shutdown within a designated grace period are then forcibly terminated with kill -9.

-n, –nodelist [nodefile]
Only the DUCC agents in the designated nodelists are shutdown. The processes are sent kill -INT signals which triggers the Java shutdown hooks and enables graceful shutdown. All user processes on the indicated nodes, both jobs and services, are sent "shutdown" signals and are given a minute to shutdown gracefully. Job and service processes which do not shutdown within a designated grace period are then forcibly terminated with kill -9.

```
stop\_ducc -n foo.nodes -n bar.nodes -n baz.nodes
```

-c, –component [component]
Stop a specific DUCC component.

This may be used to stop an errant management component and subsequently restart it (with start_ducc).

This may also be used to stop a specific agent and the job and services processes it is managing, without the need to specify a nodelist.

Examples:

Stop agents on nodes n1 and n2:

```
stop_ducc -c agent@n1 -c agent@n2
```

Stop and restart the rm:

```
stop_ducc -c rm
start_ducc -c rmc
```

Components include:

rm The Resource Manager.

or The Orchestrator.

pm The Process Manager.

sm The Service Manager.

ws The Web Server.

agentnode Node Agent on the specified node.

–nothreading If specified, the command does not run in multi-threaded mode even if it is supported on the local platform.

Notes:

Sometimes problems in the network or elsewhere prevent the DUCC components from stopping properly. The *check_ducc* command, described in the following section, contains options to query the existance of DUCC processes in the cluster, to forcibly (*kill -9*) terminate them, and to more gracefully terminate them (*kill -INT*).

12.5.3 check_ducc

Description:

Check_ducc is used to verify the integrity of the DUCC installation and to find and report on DUCC processes. It identifies processes owned by ducc (management processes, agents, and job processes), and processes started by DUCC on behalf of users.

Check_ducc can also be used to clean up errant DUCC processes when stop_ducc is unable to do so. The difference is that stop_ducc generally tries more gracefully stop processes. check_ducc is used as a last resort, or if a fast but graceless shutdown is desired.

Usage:

check_ducc [options] If no options are given this is the equivalent of:

```
check_ducc -c -n ../resources/ducc.nodes
```

This verifies the integrity of the DUCC installation and searches for all the processes owned by user *ducc* and started by DUCC on all the nodes in ducc.nodes.

Options:

-n –nodelist [nodefile] Only the nodes specified in the nodefile are searched. The option may be specified multiple times for multiple nodefiles. Note that the "local" node is always checked as well.

```
check_ducc -n nlist1 -n nlist2
```

-c –configuration Verify the Resource Manager configuration.

-p –pids Rewrite the PID file. The PID file contains the process ids of all known DUCC management and agent processes. The PID file is normally managed by start_ducc and stop_ducc and is stored in the file *ducc.pids* in directory *ducc_runtime/state*.

Occasionally the PID file can become partially or fully corrupted; for example, if a DUCC process dies spontaneously. Use check_ducc -p to search the cluster for processes and refresh the PID file.

-i, –int
Use this to send a shutdown signal (*kill -INT*) to all the DUCC processes. The DUCC processes catch this signal, close their resources and exit. Some resources take some time to close, or in case of problems, are unable to close, in which case the DUCC processes will unconditionally exit.

Sometimes problems in the network or elsewhere prevent *check_ducc -i* from terminating the DUCC processes. In this case, use *check_ducc -k*, described below.

-k, –kill
Use this to forcibly kill a component using kill -9. This should only be used if *stop_ducc* or *check_ducc -i* does not work.

-s, –single_user
Bypass the multi-user permission checks normally done on the ducc_ling utility.

–**nothreading** If specified, the command does not run in multi-threaded mode even if it is supported on the local platform.

Chapter 13

Resource Management

13.1 Overview

The DUCC Resource Manager is responsible for allocating cluster resources among the various requests for work in the system. DUCC recognizes several classes of work:

Managed Jobs Managed jobs are Java applications implemented in the UIMA framework. and are scaled out by DUCC using UIMA-AS. Managed jobs are executed as some number of discrete processes distributed over the cluster resources. All processes of all jobs are by definition preemptable; the number of processes is allowed to increase and decrease over time in order to provide all users access to the computing resources.

Services Services are long-running processes which perform some function on behalf of jobs or other services. Most DUCC services are UIMA-AS assigned to a non-preemptable resource class, as defined below.

Reservations A reservation provides non-preemptable, persistent, dedicated use of a full machine or some part of a machine to a specific user.

Arbitrary Processes An *arbitrary process* or *managed reservation* is any process at all, which may or may not have anything to do with UIMA. These processes are usually used for services, or to launch very large Eclipse workspaces for debugging. DUCC supports this type of process but is not optimized for it. These processes are usually scheduled to be non-preemptable, occupying either a dedicated machine or some portion of a machine.

In order to apportion the cumulative memory resource among requests, the Resource Manager defines some minimum unit of memory and allocates machines such that a ”fair” number of ”memory units” are awarded to every user of the system. This minimum quantity is called a share quantum, or simply, a share. The scheduling goal is to award an equitable number of memory shares to every user of the system.

The Resource Manager awards shares according to a fair share policy. The memory shares in a system are divided equally among all the users who have work in the system. Once an allocation is assigned to a user, that user’s jobs are then also assigned an equal number of shares, out of the user’s allocation. Finally, the Resource Manager maps the share allotments to physical resources.

To map a share allotment to physical resources, the Resource Manager considers the amount of memory that each job declares it requires for each process. That per-process memory requirement is translated into the minimum number of collocated quantum shares required for the process to run.

For example, suppose the share quantum is 15GB. A job that declares it requires 14GB per process is assigned one quantum share per process. If that job is assigned 20 shares, it will be allocated 20 processes across the cluster. A job that declares 28GB per process would be assigned two quanta per process. If that job is assigned 20 shares, it is allocated 10 processes across the cluster. Both jobs occupy the same amount of memory; they consume the same level of system resources. The second job does so in half as many processes however.

The output of each scheduling cycle is always in terms of processes, where each process is allowed to occupy some number of shares. The DUCC agents implement a mechanism to ensure that no user’s job processes exceed their allocated memory assignments.

Some work may be deemed to be more "important" than other work. To accommodate this, DUCC allows jobs to be submitted with an indication of their relative importance: more important jobs are assigned a higher "weight"; less important jobs are assigned a lower weight. During the fair share calculations, jobs with higher weights are assigned more shares proportional to their weights; jobs with lower weights are assigned proportionally fewer shares. Jobs with equal weights are assigned an equal number of shares. This weighed adjustment of fair-share assignments is called weighted fair share.

The abstraction used to organized jobs by importance is the job class or simply "class". As jobs enter the system they are grouped with other jobs of the same importance and assigned to a common class. The class abstraction and its attributes are described in subsequent sections.

The scheduler executes in two phases:

1. The How-Much phase: every job is assigned some number of shares, which is converted to the number of processes of the declared size.
2. The What-Of phase: physical machines are found which can accommodate the number of processes allocated by the How-Much phase. Jobs are mapped to physical machines such that the total declared per-process amount of memory does not exceed the physical memory on the machine.

The How-Much phase is itself subdivided into three phases:

1. Class counts:Apply weighed fair-share to all the job classes that have jobs assigned to them. This apportions all shares in the system among all the classes according to their weights.
2. User counts: For each class, collect all the users with jobs submitted to that class, and apply fair-share (with equal weights) to equally divide all the class shares among the users. This apportions all shares assigned to the class among the users in this class. A user may have jobs in more than one class, in which case that user's fair share is calculated independently within each class.
3. Job counts: For each user, collect all jobs assigned to that user and equally divide all the user's shares among the jobs. This apportions all shares given to this user for each class among the user's jobs in that class.

Reservations are relatively simple. If the number of shares or machines requested is available the reservation succeeds immediately. If a sufficient number of co-located shares can be made available through preemption of fair-share jobs, preemptions are scheduled and the reservation is deferred until space becomes available. resources are allocated. If space cannot be found by means of preemption, the reservation fails.

13.2 Scheduling Policies

The Resource Manager implements three coexistent scheduling policies.

FAIR_SHARE This is the weighted-fair-share policy described in detail above.

FIXED_SHARE The FIXED_SHARE policy is used to reserve a portion of a machine. The allocation is non-preemptable and remains active until it is canceled.

FIXED_SHARE allocations have several uses:

- As reservations. In this case DUCC starts no work in the share(s); the user must log in (or run something via ssh), and then manually release the reservation to free the resources. This is often used for testing and debugging.
- For services. If a service is registered to run in a FIXED_SHARE allocation, DUCC allocates the resources, starts and manages the service, and releases the resource if the service is stopped or unregistered.
- For UIMA jobs. A "normal" UIMA job may be submitted to a FIXED_SHARE class. In this case, the processes are never preempted, allowing constant and predictable execution of the job. The resources are automatically released when the job exits.

Note: A fixed-share request specifies a number of processes of a given size, for example, "10 processes of 32GB each". The ten processes may or may not be collocated on the same machine. Note that the resource manager attempts to minimize fragmentation so if there is a very large machine with few allocations, it is likely that there will be some collocation of the assigned processes.

Note: A fixed-share allocation may be thought of a reservation for a "partial" machine.

RESERVE The RESERVE policy is used to reserve a full machine. It always returns an allocation for an entire machine. The reservation is permanent (until it is canceled) and it cannot be preempted by any other request.

Reservations may also be used for services or UIMA jobs in the same way as FIXED_SHARE allocations, the difference being that a reservation occupies full machines and FIXED_SHARE occupies portions of a machine.

Note: It is possible to configure the scheduling policy so that a reservation returns any machine in the cluster that is available, or to restrict it to machines of the size specified in the reservation request.

13.3 Priority vs Weight

It is possible that the various policies may interfere with each other. It is also possible that the fair share weights are not sufficient to guarantee sufficient resources are allocated to high importance jobs. Priorities are used to resolve these conflicts

Simply: priority is used to specify the order of evaluation of the job classes. Weight is used to proportionally allocate the number of shares to each class under the weighted fair-share policies.

Priority. It is possible that conflicts may arise in scheduling policies. For example, it may be desired that reservations be fulfilled before any fair-share jobs are scheduled. It may be desired that some types of jobs are so important that when they enter the system all other fair-share jobs be evicted. Other such examples can be found.

To resolve this, the Resource Manager allows job classes to be prioritized. Priority is used to determine the order of evaluation of the scheduling classes.

When a scheduling cycle starts, the scheduling classes are ordered from "best" to "worst" priority. The scheduler then attempts to allocate ALL of the system's resources to the "best" priority class. If any resources are left, the scheduler proceeds to schedule classes in the next best priority, and so on, until either all the resources are exhausted or there is no more work to schedule.

It is possible to have multiple job classes of the same priority. What this means is that resources are allocated for the set of job classes from the same set of resources. Resources for higher priority classes will have already been allocated, resources for lower priority classes may never become available.

To constrain high priority jobs from completely monopolizing the system, class caps may be assigned. Higher priority guarantees that some resources will be available (or made available) but doesn't require that all resources necessarily be used.

Weight. Weight is used to determine the relative importance of jobs in a set of job classes of the same priority when doing fair-share allocation. All job classes of the same priority are assigned shares from the full set of available resources according to their weights using weighted fair-share. Weights are used only for fair-share allocation.

13.4 Node Pools

It may be desired or necessary to constrain certain types of resource allocations to a specific subset of the resources. Some nodes may have special hardware, or perhaps it is desired to prevent certain types of jobs from being scheduled on some specific set of machines. Nodepools are designed to provide this function.

Nodepools impose hierarchical partitioning on the set of available machines. A nodepool is a subset of the full set of machines in the cluster. Nodepools may not overlap. A nodepool may itself contain non-overlapping subpools. It is possible to define and schedule work to multiple, independent nodepools.

Job classes are associated with nodepools. During scheduling, a job may be assigned resources from its associated nodepool, or from any of the subpools which divide the associated nodepool. The scheduler attempts to fully exhaust resources in the associated nodepool before allocating within the subpools, and during eviction, attempts to first evict from the subpools. The scheduler ensures that the nodepool mechanism does not disrupt fair-share allocation.

More information on nodepools and their configuration can be found here.

13.5 Job Classes

The primary abstraction to control and configure the scheduler is the class. A class is simply a set of rules used to parametrize how resources are assigned to jobs. Every job that enters the system is associated with a single job class.

The job class defines the following rules:

Priority This is the order of evaluation and assignment of resources to this class. See the discussion of priority vs Weight for details.

Weight This defines the "importance" of jobs in this class and is used in the weighted fair-share calculations.

Scheduling Policy This defines the policy, fair share, fixed share, or reserve used to schedule the jobs in this class.

Cap Class caps limit the total resources assigned to a class. This is designed to prevent high importance and high priority job classes from fully monopolizing the resources. It can be used to limit the total resources available to lower importance and lower priority classes.

Nodepool A class may be associated with exactly one nodepool. Jobs submitted to the class are assigned only resources which lie in that nodepool, or in any of the subpools defined within that nodepool.

Prediction For the type of work that DUCC is designed to run, new processes typically take a great deal of time initializing. It is not unusual to experience 30 minutes or more of initialization before work items start to be processed.

When a job is expanding (i.e. the number of assigned processes is allowed to dynamically increase), it may be that the job will complete before the new processes can be assigned and the work items within them complete initialization. In this situation it is wasteful to allow the job to expand, even if its fair-share is greater than the number of processes it currently has assigned.

By enabling prediction, the scheduler will consider the average initialization time for processes in this job, current rate of work completion, and predict the number of processes needed to complete the job in the optimal amount of time. If this number is less than the job's fair, share, the fair share is capped by the predicted needs.

Prediction Fudge When doing prediction, it may be desired to look some time into the future past initialization times to predict if the job will end soon after it is expanded. The prediction fudge specifies a time past the expected initialization time that is used to predict the number of future shares needed.

Initialization cap Because of the long initialization time of processes in most DUCC jobs, process failure during the initialization phase can be very expensive in terms of wasted resources. If a process is going to fail because of bugs, missing services, or any other reason, it is best to catch it early.

The initialization cap is used to limit the number of processes assigned to a job until it is known that at least one process has successfully passed from initialization to running. As soon as this occurs the scheduler will proceed to assign the job its full fair-share of resources.

Expand By Doubling Even after initialization has succeeded, it may be desired to throttle the rate of expansion of a job into new processes.

When expand-by-doubling is enabled, the scheduler allocates either twice the number of resources a job currently has, or its fair-share of resources, whichever is smallest.

Maximum Shares This is for FIXED_SHARE policies only. Because fixed share allocations are not preemptable, it may be desirable to limit the number of shares that any given request is allowed to receive.

Enforce Memory This is for RESERVE policies only. It may be desired to allow a reservation request receive any machine in the cluster, regardless of its memory capacity. It may also be desired to require that an exact size be specified (to ensure the right size of machine is allocated). The enforce memory rule allows installations to create reservation classes for either policy.

More information on nodepools and their configuration can be found here.

Chapter 14

Simulation and System Testing

This chapter describes the large-scale testing and cluster-simulation tools supplied with DUCC. This is of use mostly to contributors and developers of DUCC itself.

DUCC is shipped with support for simulating large clusters of arbitrarily configured nodes. A simple control file describes some number of simulated nodes of arbitrary memory sizes. DUCC's design allows multiples of these to be spawned on a single node, or on a small set of nodes with multiple simulated nodes apiece. The standard testing configuration used for most of the development of DUCC consisted of four physical 32-GB machines running 52 simulated nodes of varying memory sizes from 32 to 128-GB each.

To simulate job loads, a simple UIMA-AS job that sleeps for some easily configured length of time was constructed. Another control file is used to generate job specifications requesting randomly-chosen job parameters such as memory requirements, service dependencies, scheduling classes, and so on.

The test suite contains a simple UIMA Analysis Engine called `FixedSleepAE`, and a simple Collection Reader called `FixedSleepCR`. The CR reads a set of sleep times, creates CASs, and ships them to the AEs via DUCC's Job Driver. The CAS contains the time to sleep and various parameters regarding error injection.

The AE receives a CAS, performs error injection if requested, and sleeps the indicated period of time, simulating actual computation but requiring very few physical resources. Hence, many of these may be run simultaneously on relatively modest hardware.

Developers may construct arbitrary jobs by creating a file with sleep times designed to exercise what ever is necessary. DUCC ships with the three primary job collections (test suites) used during initial development. The suites are based on actual workloads and have shown to be very robust for proving the correctness of the DUCC code under stress.

The cluster simulator has been also been run on a 4GB iMac with 8 simulated Agents, an 8GB MacBook with the same configuration, a 32GB iMac with up to 40 simulated Agents. It has also been scaled up to run on 8 45GB Intel nodes running Linux, simulating 20TB of memory.

The rest of this chapter describes the mechanics of using these tools.

14.1 Cluster Simulation

14.1.1 Overview

Cluster-based tools such as DUCC are very hard to test and debug because all interesting problems occur only when the system is under stress. Acquisition of a cluster of sufficient size to expose the interesting problems is usually not practical.

DUCC's design divorces all the DUCC processes from specific IP addresses or node names. ActiveMQ is used as a nameserver and packet router so that all messages can be delivered by name, irrespective of the physical hardware the destination process may reside upon.

A DUCC system is comprised of three types of processes (daemons):

1. The DUCC management daemons:
    - The Orchestrator (OR). This is the primary point of entry to the system and is responsible for managing the life cycle of all work in the system.
    - The Process Manager (PM). This is responsible for managing message flow to and from the DUCC Agents.
    - The Resource Manager (RM). This is responsible for apportioning system resources among submitted work (jobs, reservations, services).
    - The Service Manager (SM). This is responsible for keeping services active and available as needed.
    - The Web Server (WS). This process listens to all the state messages in the system to provide a coherent view of DUCC to the outside world.
2. The DUCC Node Agents, or simply, Agents. There is one Agent running on every physical node.
3. The ActiveMQ Broker. All message flow in the system is directed through the ActiveMQ broker, with the exception of the CLI, (which uses HTTP).

Normally, the DUCC Agents report the name, IP address, and physical memory of the node they actually do reside upon. This is simply for convenience. It is possible to parametrize the DUCC Agents to report any arbitrary name and address to the DUCC. DUCC components that need to know about Node Agents establish subscriptions to the Agent publications with ActiveMQ and build up their internal structures from the node identities in the Agent publications. Processes which normally establish agent listeners are are the RM, PM, and WS.

It is also possible to parametrize a DUCC agent to cause it to report any arbitrary memory size. Thus, an agent running on a 2GB machine can be started so that it reports 32GB of memory. This parametrization is specifically for testing, of course.

The ability to parametrize agent identities and memory sizes is what enables cluster simulation. A control file is used by start-up scripting to spawn multiple agents per node, each with unique identities.

14.1.2 Node Configuration

A Java properties file is used to configure simulated nodes. There are three types o entries in this file:

nodes This single entry provides the blank-delimited names of the physical nodes participating in the simulated cluster.

memory This single line consists of a blank-delimited set of numbers. Each number corresponds to some memory size, in GB, to be simulated.

node descriptions There are one or more of these. The format of each line is

```
[nodename].[memory] = [count]
```

where

nodename is the name of one of the nodes in the *nodes* line mentioned above.

memory is one of the memory sizes given in the *memory* line mentioned above.

count is the number of simulated agents in the indicated node, with the indicated memory, to be simulated.

For example, the following simulated cluster configuration defines twenty (20) simulated nodes, all to be run on the single physical machine called *agentn*. The simulated nodes contain a mix of 31GB, 47GB, and 79GB memory sizes. There are 7 31GB nodes, 7 47GB nodes, and 6 79GB nodes.

```
# names of nodes in the test cluster
nodes       = agentn
```

```
# set of memory sizes to configure
memory      = 31 47 79

# how to configure memories: node.memsize = count
agentn.31 = 7
agentn.47 = 7
agentn.79 = 6
```

The nodenames generated by this means are the name of the physical node where the agent is spawned, and a numeric id appended, for example,

```
  agentn-1
  agentn-2
  agentn-3
    etc.
```

14.1.3 Setting up Test Mode

During simulation and testing it is desirable and usually required that DUCC run in unprivileged mode, with all processes belonging to a single userid. Unfortunately, this does not exercise any of the multi-user code paths, especially in the Resource Manager.

To accommodate this, DUCC can be configured to run in "test mode", such that work is submitted under "simulated" userid which DUCC treats as discrete IDs. All actual work is executed under the ownership of the tester however.

To establish test mode:

1. Ensure that *ducc.properties* is configured to point to a non-privileged version of *ducc_ling*. Specifically, configure this line in *ducc.properties*

   ```
   ducc.agent.launcher.ducc_spawn_path=/home/ducctest/duccling.dir/amd64/ducc_ling
   ```

   in this example a version of *ducc_ling* known to not have elevated privileges it configured.

2. Configure test mode in *ducc.properties*:

   ```
   ducc.runmode=Test
   ```

   IMPORTANT: Do not start DUCC with *ducc.runmode=Test* if *ducc_ling* has elevated privileges. Test mode bypasses the authentication and authorization checks that are normally used and the system would run completely open.

In test mode, jobs may specify what simulated userid is to be used. Most of DUCC does not pay any attention to the user so this works fine, and the parts that do care about the user are bypassed when *ducc.runmode=Test* is configured.

14.1.4 Starting a Simulated Cluster

DUCC provides a start-up script in the directory `$DUCC_HOME/examples/systemtest` called `start_sim`.

WARNING: Cluster simulation is intended for DUCC testing, including error injection. It is similar to flying a high-performance fighter jet. It is intentionally twitchy. Very little checking is done and processes may be started multiple time regardless of whether is sane to do this.

To start a simulated cluster, use the *start_sim* script:

Description: The *start_sim* script is used to start a simulated cluster.

Usage: *start_sim* options

Options:

-n, –nodelist [nodelist] where the nodelist is a cluster description as described above.

-c –components [component list] . The component list is a blank-delimited list of components including *or, rm, sm, pm, ws, broker* to start an individual component, or *all* to start all of the components. NOTE: It is usually an error to start any of these components more than once. However *start_sim* allows it, to permit error injection.

–nothreading If specified, the command does not run in multi-threaded mode even if it is supported on the local platform.

14.1.5 Stopping a Simulated Cluster

There are two mechanisms for stopping a simulated cluster:

1. *check_ducc -k* This looks for all DUCC processes on the nodes in $DUCC_HOME/resources/ducc.nodes and issues *kill -9* to each process. It then removes the Orchestrator lock file. This is the most violent and surest way to stop a simulated DUCC cluster. In order for this to work, be sure to include the names of all physical nodes used in the simulated cluster in the DUCC configuration file *$DUCC_HOME/resources/ducc.nodes.* It is described in the administration section of the book.

2. *stop_sim* With no arguments, this attempts to stop all the simulated agents and the management daemons using *kill -INT*. It is possible to stop individual agents or management nodes by specifying their component IDs. The kill signals *-KILL, -STOP* and *-CONT* are all supported. This allows error injection as well as a more orderly shutdown than *check_ducc -k*.

Note that *check_ducc* is found in *$DUCC_HOME/admin.* The *stop_sim* script is found in *duccruntime/examples/systemtest.*

The *start_sim* script creates a file called *sim.pids* containing the physical node name, Unix process ID (PID), and component ID (ws, sm, or, pm, rm) of each started DUCC component. In the case of agents, each agent is assigned a number as a unique id. These ids are used with *stop_sim* to affect specific processes. If the cluster is stopped without using *stop_sim*, or if it simply crashes, this PID file will get out of date. Fly more carefully next time!

stop_sim works as follows:

Description The *stop_sim* script is used to stop some or all of a simulated cluster.

Usage: *stop_sim* [options]

Options:

-c, –component [component name] where the name is one of *rm, sm, pm, or. ws,. Kill -INT* is used to enable orderly shutdown unless overridden with -k, -p, or -r as described below.

-i, –instance [instance-id] where the instance-id is one of the agent ids in “sim.pids”. *Kill -INT* is used to enable orderly shutdown unless overridden with -k, -p, or -r as described below.

-k, –kill Use *kill -9* to kill the process.

-p, –pause Signal the process with *SIGSTOP*.

-r, –resume Signal the process with *SIGCONT*.

–nothreading If specified, the command does not run in multi-threaded mode even if it is supported on the local platform.

14.2 Job Simulation

14.2.1 Overview

“Real” jobs are highly memory and CPU intensive. For testing and simulation purposes, the jobs need not use anywhere close to their declared memory, and need not consume any CPU at all. The FixedSleepAE is a UIMA analytic that is given a time, in milliseconds, and all it does is sleep for that period of time and then exit. By running many of these in a simulated cluster it is possible to get all the DUCC administrative processes to behave as if there is a real load on the system when in fact all the nodes and jobs are taking minimal resources.

The FixedSleepAE is delivered CASs by the FixedSleepCR. This CR reads a standard Java properties file, using the property “elapsed” to derive the set of sleep times. On each call to the CR’s “getNext()” method, the next integer from “elapsed” is fetched, packaged into a CAS, and shipped to ActiveMQ where it is picked up by the next available FixedSleepAE.

The test driver is given a control file with the names of all the jobs to be submitted in the current run, and the elapsed time to wait between submission of each job. Each job name corresponds to a file that is not an actual DUCC specification, but rather the description of a DUCC specification. Each description is a simple Java properties file.

To submit a job, the test driver reads the next job description file derive the number of threads, the simulated user, the desired (simulated) memory for the job, (possibly) the service ID, and the scheduling class for the job. From these it constructs a DUCC job specification and submits it to DUCC.

Scripting is used to read the job meta-descriptors and generate a control file that submits the job set with a large set of variations. The same scripting reads each meta-descriptor and modifies it according to the specific parameters of the run, adjust things such as scheduling class, memory size, etc.

14.2.2 Job meta-descriptors

For each simulated job in a run, a meta-descriptor must be constructed. These may be constructed “by hand”, or via local scripting, for example from log analysis. (The packaged meta-descriptors are generated from logs of actual workloads.)

A meta-descriptor must contain the following properties:

tod This specifies a virtual “time of day of submission”, starting from time 0, specified in units of milliseconds, when the job is to be submitted. During job generation, this may be used to enforce precise timing of submission of the jobs.

elapsed This is a blank-delimited set of numbers. Each number represents the elapsed time, in milliseconds, for a single work item. There must be one time for each work item. These numbers are placed into CASs by the job’s Job Driver and delivered to each Job Process. For example, if this job is to consist of 5 work items of 1, 2, 3, 4 and 5 seconds each, specify

```
elapsed = 1000 2000 3000 4000 5000
```

threads This is the number of threads per Job Process. It is translated to the *process_thread_count* parameter in the job specification.

user This is the name of the user who owns the job. It may be any string at all. If DUCC is started in *test* mode, this will be shown as the owner of the job in the webserver and the logs.

memory This is the amount of memory to be requested for the job, translating to the job specification’s *process_memory_size* parameter.

class This is the scheduling class for the job.

machines This is the maximum number of processes to be allocated for the job, corresponding to the *process_deployments_max* parameter.

For example:

```
tod = 0
elapsed = 253677 344843 349342 392883 276264 560153 162850 744822 431210 91188 840262 843378
threads = 4
user = Rodrigo
memory = 20
class = normal
machines = 11
```

All the job meta-descriptors for a run must be placed into a single directory.

14.2.3 *Prepare* Descriptors

A *prepare descriptor* is also a standard Java properties file. This defines where the set of meta descriptors resides, where to place the modified meta-files, how to assign scheduling classes to the jobs, how to apportion memory sizes, how to apportion services, how long the total run should last, and how to compress sleep times.

All parts of the run are randomized, but the randomization can be made deterministic between runs by specifying a seed to the random number generator.

Properties include

random.seed This is the random-number generator seed to be used for creating the run.

src.dir This is the directory containing the input-set of meta-specification files.

dest.dir This is the directory that will contain the updated meta-specification files.

scheduling.classes This is a blank-delimited list of the scheduling classes to be randomly assigned to the jobs.

scheduling.classes.[name] Here, *name* is the name of one of the scheduling classes listed above. The value is a weight, to be used to affect the distribution of scheduling classes among the jobs.

job.memory This is a blank-delimited list of memory sizes to be randomly assigned to each job.

job.memory.[men]] Here, *mem* is one of the memory sizes specified above. The value is a weight, used to affect the distribution of memory sizes among the jobs.

job.services This is a blank-delimited list of a service id, where the id is one of the services specified in the *services.boot* control file.

job.services.[id] Here *id* is one of the ids specified in the job.services line above. The value is a weight, used to affect the distribution of services among the jobs.

submission.spread This is the time, in seconds the set of job submissions is to be spread across. The jobs are submitted at random times such that the total time between submitting the first job and the last job is approximately this number.

compression For each sleep time in the job, divide the actual value by this number. This allows testers to use the actual elapsed time from real jobs, and compress the total run time so it fits approximately into the submission spread.

For example, if a collection of jobs was originally run over 24 hours, but you want to run a simulation with approximately the same type of submission that last only 15 minutes, specify a submission spread of 900 (15 minutes) and a compression of 96.

Here is a sample run configuration file:

```
# control file to create a random-like submission of jobs for batch submission
# This represents jobs submitted over approximately 36 hours real time
# Compression of 96 and spread 920 gives a good 15-20 minute test on test system with
# 136 15GB shares

random.seed                   = 0         # a number, for determinate randoms
```

```
                                          # or TOD, and the seed will use
                                          # current time of day

src.dir                       = jobs.in   # where the jobs are
dest.dir                      = jobs      # where to put prepared jobs

scheduling.classes            = normal    # classes
scheduling.classes.normal     = 100

job.memory                    = 28 37     # memorys to assign
job.memory.28                 = 50
job.memory.37                 = 50

job.services                  = 0 1 2 3 4 5 6 7
job.services.0                = 25
job.services.1                = 25
job.services.2                = 25
job.services.3                = 25
job.services.4                = 25
job.services.5                = 25
job.services.6                = 25
job.services.7                = 25

submission.spread             = 920       # number of *seconds* to try to spread submission over

compression                   = 96        # comporession for timings
```

14.2.4 Services

It is possible to run the FixedSleepAE as a UIMA-AS service, with each job specifying a dependency on the service, and the indicated service doing the actual sleeping on behalf of the job.

These variants on services are supported:

1. Registered services, started by reference,
2. Registered services, started by the simulator,

To use these simulated services, configure a "service boot" file and reference the services from the job generation config file.

Properties required in the service boot file include:

register This specifies registered services. The value is a blank delimited list of pseudo IDs for the registered services.

start This specifies which of the registered services to automatically start. The value is some subset of the pseudo IDS specified under *register*

instances_[id] Here *id* is one of the IDs specified for *submit, register,* or *standalone.* The value is the number of instances of that specific service to set up.

Service pseudo IDs DUCC is packaged with 10 pre-configured services that use the FixedSleepAE. All of these services behave identically, the only difference is their endpoints, which allows the simulated runs to activate and use multiple independent services. Because the endpoints are in the various UIMA XML service descriptors, it is necessary to use exactly these IDs when generating a test run. Thus, the only valid pseudo-ids for service configuration are *0, 1, 2, 3, 4, 5, 6, 7, 8, 9.*

These *service ids* are used on the job configuration file to establish a weighted distribution of service use among the jobs.

Here is a sample service configuration file:

```
# register these services, 2 instances each
register 0 1 2 3
instances_0 2
instances_1 2
instances_2 2
instances_3 2

# start these registered services
start 2 3
```

14.2.5 Generating a Job Set

The *prepare* script, found in $DUCC_HOME/examples/systemtest is used to generate a test run from the control files described above. To use it, execute

```
prepare [config-file]
```

where *config file* is the run description file described above.

This script reads the meta-specification in the *jobs.in* directive of the config-file, generates a set of meta-specification files into the *jobs.out* directory, and creates a control file, *job.ctl*. The *job.ctl* file is used by the simulation driver to submit all the jobs.

14.2.6 Running the Test Driver

A test run is driven from the script *runducc* which resides in the directory *$DUCC_HOME/examples/systemtest*. This script supports a large number of options intended to inject errors and otherwise perturb a run.

To use the test driver, first create a job collection as described above. This will generate a file called *job.ctl* in the test directory containing the *prepare* file.

Then execute:

```
    runducc -d jobdir -b batchfile options...
```

where the various parameters and options include:

-d jobdir The jobdir is the directory containing the *prepare* file and the *job.ctl* file as describe in the previous section.

-b batchfile The batchfile is usually *job.ctl* as generated by the prepare script. (This file may be hand-edited to create custom runs outside of the *prepare* script.)

–**AE** This specifies to run all jobs as CR and AE. This is the default and need not be specified.

–**DD** This specifies to run all jobs as CR and DD. The jobs are generated as DD-style jobs, as opposed to AE.

–**SE cfg** This specifies to run all jobs using services, as generated by the *prepare* script. The parameter is the service config file as described above. When specified, the driver starts the services as configured, pauses a bit to let them start up, and generated every job with a dependency on one of the services.

-i time-in-sec If specified, this forces each AE to spend a minimum of the indicated time in it's initialization method (also a sleep). If not specified, the default is 10 seconds. The actual time is controlled by the *-r* (range) option.

–**init_fail_cap count** This sets the job property *process_initialization_failures_cap* to the the indicated value, to control the number of initialization failures to be tolerated before terminating the job.

–int_timeout seconds This sets the job property *process_initialization_time_max* to the indicated value, to control the time allowed in initialization before failure is reported.

-r time-in-sec This specifies the top range for initialization. The service will spend the time specified in *-i*, PLUS a random value from 1 to the time specified in *-r* in its initialization phase.

–IB The Job Process will leak memory in it's initialization phase until it is killed, hopefully by DUCC, but possibly by the operating system. *Use with care..*

–PB The job Process will leak memory in it's processing phase until it is killed, hopefully by DUCC, but possibly by the operation system. *Use with care.*

-m size-in-gb Memory override. Use this value for all jobs, overriding the value in the generated meta-specification file.

-n max-Number-of-processes Max machine override. If specified, this overrides the configured process max from the job control file. Specify the max as 0 and no maximum will be submitted with the job, causing the scheduler to try to allocated the largest possible number of processes to the job.

-p time-in-seconds If specified the job property *process_per_item_time_max*, which sets a timeout on work items, is set to the indicated time.

-w, –watch Submit every job with the *wait_for_completion* flag. This runs the driver in multi-threaded mode, with each thread monitoring the progress of a job.

-x rate This specifies an expected error rate for execution phase in a job process, from 0-100 (a percentage). When specified, each job process uses a random number generator to determine the probability that is would crash, if if that probability is within the specified rate, it generates a random exception.

-y rate This specifies an expected error rate for initialization phase in a job process, from 0-100 (a percentage). When specified, each job process uses a random number generator to determine the probability that is would crash, if if that probability is within the specified rate, it generates a random exception.

For an expected error-free run, only the -b and -d options are needed.

14.3 Pre-Packaged Tests

Three test suites are provided using the mechanisms described in the previous section:

- A 15-minute run comprising approximately 30 jobs. This includes configuration for single-class submission, mixed class submission, and one configured to maximize resource fragmentation.
- A 30-minute run comprising approximately 33 jobs. This includes a single configuration.
- a 24-hour run comprising approximately 260 jobs. This also includes configurations for single-class submission, mixed classes, and fragmentation. *Note: this run has been reconfigured to run in 12 hours, and has been successfully been configured to complete in 6 hours. This can create a significant load on the DUCC processes.*

The configurations are found in the $DUCC_HOME/examples/systemtest directory and are in sub directories called,

- mega-15-min
- mega-30-min
- mega-24-hour

To run these tests:

1. Create a node configuration. A sample configuration to generate 52 simulated nodes, and which assumes the physical machines for the simulation are called *sys290, sys291, sys292, sys293* and *sys534* is supplied in $DUCC_HOME/examples/systemtest. Change the node names to the names of real machines, making any other adjustments needed.

2. Update your *$DUCC_HOME/resources/ducc.nodes* so that all the real node names specified int the simulated node file are included.

3. Update your *$DUCC_HOME/resources/ducc.properties* so the *ducc.head* is specified as the *real, physical* machine where you will start the simulated cluster.

4. Be sure the *job driver* nodepool, if configured in *$DUCC_HOME/resources/ducc.classes*, specifies the name of one of the simulated nodes. When first running these tests it is usually best that the job driver NOT be configured on a specific node in *ducc.classes* as it can be confusing to get this right on simulated clusters.

   Specifically, in *ducc.classes*, configure the *JobDriver* class thus:

   ```
     Class JobDriver fixed-base { }
   ```

   This allows DUCC to schedule the job driver on any node in the simulated cluster.

5. Generate the job set. For example, to generate the job set for the 15-minute run,

   ```
   cd $DUCC_HOME/examples/systemtest
   ./prepare mega-15-min/jobs.prepare
   ```

6. Start the simulated cluster (Assuming your simulated node file is called *52.simulated.nodes*:

   ```
   cd $DUCC_HOME/examples/systemtest
   ./start_sim -c all -n 52.simulated.nodes
   ```

7. Use the webserver (or for advanced users, log files), to ensure everything came up and the job driver node has been assigned.

8. Start the run:

   ```
   cd $DUCC_HOME/examples/systemtest
   ./runducc -d mega-15-min -b job.ctl
   ```

Part V

Ducc Principles of Operation

Chapter 15

Platform

The Distributed UIMA Cluster Computing (DUCC) platform comprises software designed to facilitate the scale-out of Unstructured Information Management Architecture (UIMA) pipelines on a collection of nodes (machines, computers) shared ”fairly” by a group of users.

The major components of DUCC are the Orchestrator (OR), the JobDriver (JD), the Resource Manager (RM), the Process Manager (PM), the Services Manager (SM), the Agents, the Command Line Interface (CLI), the Application Program Interface (API), and the WebServer (WS).

15.1 Highlights

DUCC was conceived to address the following:

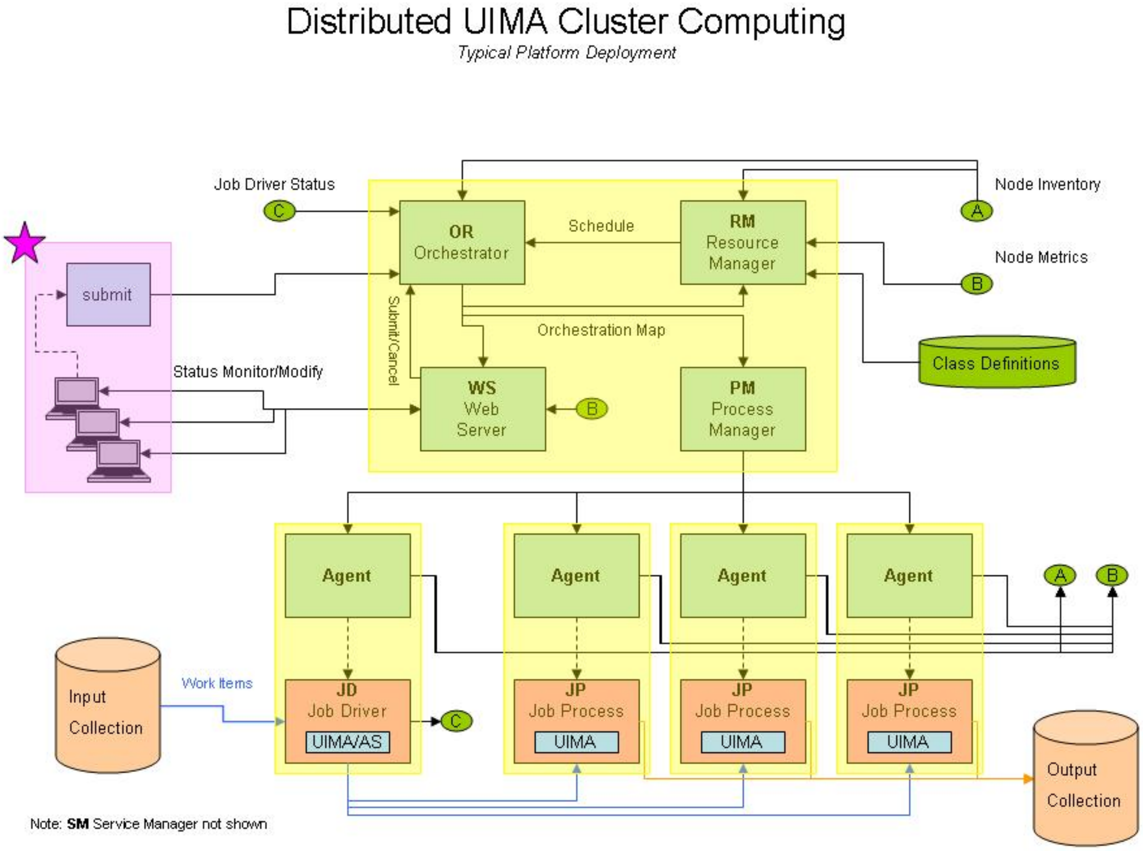
- manage a cluster of machines for UIMA workloads
- highly configurable ”fair-share” resource allocation system
- application code runs with credentials of submitting user
- ”virtual machine” resources for user processes allocated instantaneously via Linux Control Groups
- extensive Web, CLI and API interfaces
- rich debugging support for user processes

15.2 Architecture

The DUCC platform employs building-block software from the open-source community where possible to achieve it goals. Foremost and not surprisingly DUCC employs in its foundation UIMA-AS, which in-turn relies upon UIMA-Core.

Additionally, Camel is used for inter-component communications. ActiveMQ is employed to process work items amongst a distributed set of work item processors. Logging is facilitated by Log4J. Jetty is used for the WebServer and jQuery is deployed to web browsers. And various other open-source softwares are likewise employed.

By employing reliable open-source code where possible, the amount of custom code needed to develop and maintain DUCC functionality is minimized. And substitution of implementation for equivalent functionality is possible, for example replacing Apache Active MQ with IBM WebSphere MQ.



15.3 Jobs

The main focus of the system is running ”batch” Jobs comprising UIMA pipelines.

Users submit Jobs to the system to be deployed and executed. Jobs have a life-cycle from birth to death during which time a (normally) finite collection of work items are processed by one or more UIMA pipelines. Jobs consist of two parts: a singleton work item supplier, known in UIMA parlance as a *Collection Reader* (CR); and one or more pipelines, each known in UIMA parlance as an *Analysis Engine* (AE).

15.3.1 Characteristics

DUCC facilitates ”fair-share” UIMA pipeline scale-out.

The UIMA pipelines comprising a Job represent ”embarrassingly parallel” deployments. Over time, a Job may expand and contract with respect to the number of UIMA pipelines deployed during its lifetime. This may be due to the introduction or completion of other Jobs, the rise and fall of other resource consumers such as Reservations or Services, and the addition or removal of computer resources to the cluster.

With respect to contraction, each UIMA pipeline must be prepared to process work items that may have been partially processed previously.

Pipelines themselves may comprise one or more duplicate threads, such that each pipeline can simultaneously process multiple work items. The number of pipelines and threads per pipeline are configurable per Job.

15.3.2 Performance

For the distributed environment, DUCC relies upon a Network File System (NFS) for file access to work items. High performance is achieved through NFS data sharing and (via ActiveMQ) the passing of data-handles that are utilized by the ”embarrassingly parallel” pipelines.

15.4 Reservations

To help support Jobs, DUCC provides facilities for Reservations of two types: Managed and Unmanaged. Reservations, once allocated, are preserved until canceled.

Managed Reservations (MRs) comprise ”arbitrary” processes, for example Java programs, c-programs, bash shells, etc.

Unmanaged Reservations (URs) comprise a resource that can be utilized for any purpose, subject to the limitations of the assigned *DUCC-Share* or *DUCC-Shares*.

15.5 Services

To help support Jobs, DUCC provides facilities for Services of two types: UIMA and Ping-Only. Services can be predefined in a registry, and Jobs can declare dependency on one or more of them.

Services can be shared by my multiple Jobs or can be tied to just one. Services can be started at DUCC-boot time or at Service-definition time or at Job launch time.

Services can be expanded and contracted by command or on-demand. Services can be stopped by command or due to absence of demand.

Services nominally exists for reasons of efficiency due to high start-up costs or high resource consumption. Benefits of cost amortization are realized by sharing Services amongst a collection of Jobs rather than employing a private copy for each.

The lifecycle of each UIMA Service is managed by DUCC, which is not the case for Ping-Only Services. However, each comprises a ”pinger” which adheres to a standard interface and provides health and statistical information.

15.6 Management

The DUCC system employs several management techniques to fairly apportion resources.

15.6.1 Memory Shares

The DUCC system partitions the entire set of available resources comprising nodes (machines, computers) into *DUCC-Shares*.

Partitioning of the available nodes (machines, computers) into *DUCC-Shares* facilitates multitenancy amongst a collection of DUCC-managed user applications consisting of UIMA pipelines.

One or more *DUCC-Shares* are allocated and sub-partitioned into *JD-Shares*.

Users submit Jobs to the DUCC system specifying a requisite memory size. Each Job is allocated one *JD-Share* and, based upon user specified memory size, one or more *DUCC-Shares*. Likewise, users submit Reservations and Services also comprising memory size information. These are assigned *DUCC-Shares* only.

New Jobs, Reservations and Services may only enter the system when there are sufficient unallocated *DUCC-Shares* available. To make room for newly arriving submissions, the Resource Manager may preempt use of already previously assigned *DUCC-Shares* for re-assignment.

15.6.2 Linux Control Groups

If available, DUCC employs Linux Control Groups to enforce limits on deployed applications. Exceeding limits penalizes only the offender. For example, if a user application exceeds its memory *DUCC-Share* size then it is forced to swap while other co-resident applications remain unaffected.

15.6.3 Preemption

Preemption is employed by DUCC to re-apportion *DUCC-Shares* when new work is submitted.

For example, presume a simple DUCC system with just one preemptable scheduling class and resources comprising 11 *DUCC-Shares*. Further, suppose that 1 *DUCC-Share* is allocated for partitioning into *JD-Shares*. When the Job #1 is submitted it is entitled to all remaining 10 shares. When Job #2 arrives, each job is entitled to only 5 shares. Thus, 5 *DUCC-Shares* from Job #1 are preempted and reassigned to Job #2.

Chapter 16

System Organization

16.1 Single System Image

DUCC runs on Linux. It can be run on a single system in simulation-mode or on a cluster (two or more machines). For clusters, DUCC replies upon these requirements:

- common userids across the cluster

  Each userid must have the same definition on all machines participating in the DUCC cluster.

- a shared filesystem for user and DUCC data across the cluster

  Each machine shares a filesystem (commonly provided by NFS) with all machines participating in the DUCC cluster.

16.2 Communications

DUCC comprises a collection of singleton and distributed daemons that need to coordinate activities. This coordination is accomplished via messaging.

The system is fault tolerant with respect to lost messages, since publications occur at regular intervals and each message encapsulates the current and/or desired state for the target audience. As such, actions may be be delayed but will be carried out as soon as the next message arrives.

16.3 Daemons

DUCC is implemented through a collection of configurable singleton and distributed daemons.

16.3.1 Orchestrator (OR)

There is one Orchestrator per DUCC cluster.

The duties of the Orchestrator are:

receive and act upon user submitted application requests;

manage and publish common state to a set of distributed components;

maintain checkpoint and historical state;

manage the lifecycle of jobs, services, and reservations.

The Orchestrator provides essential functionality for operation of the DUCC system. It is configurable and tunable.

The Orchestrator receives user requests to start, stop and modify Jobs, Services and Reservations. It manages the life-cycles of these entities, each deployed to a managed cluster of machines (nodes, computers).

The Orchestrator both publishes and receive reports. The Orchestrator publication is also known as the `OR-map`, which is the final authority on the state of Jobs, Reservations, and Services. All other DUCC components respect the state published by the Orchestrator and each carries out its assigned duties accordingly.

Controller

The Orchestrator Controller responsibilities entail receiving user submitted requests and processing them to completion in accordance with an instance of the appropriate state machine.

User submitted requests comprise:

- Job { Start, Stop, Modify }
- Reservation { Start, Stop, Modify }
- Service { Start, Stop, Modify }
- Individual Process { Stop }

The Controller responsibilities further entail receiving status messages from other DUCC components and advancing the state machines of user submitted Jobs, Reservations, and Services as necessary.

Additionally and importantly, the Controller is the final authority for the DUCC system state comprising active Jobs, Reservations, and Services. The Controller publishes DUCC system state at regular intervals for consumption and use by all other DUCC components.

Authenticator

The authenticator determines whether or not the requesting user is a DUCC administrator. Such users have special privileges, such as:

- the ability to control DUCC system functions
- the ability to act on behalf of other users

The file `ducc.administrators` comprises the list of privileged DUCC users.

Validation

Each request to submit, cancel or modify is validated against a set of criteria that define acceptableness. In the case of missing information, a default value may be employed.

Presently, the following keys are validated by the Orchestrator:

- `process_thread_count`
- `number_of_instances`
- `scheduling_class`

Other keys are validated by the Command Line Interface, prior to arrival at the Orchestrator.

Factory

Once accepted, submit requests proceed through a corresponding factory to have a state machine representation entered into the published `OR-map` with initial state of `Received`. The request remains active until it advances to final state `Completed`.

Each factory-created representation comprises appropriate information as follows:

- - standard information
    - unique identifier (assigned by DUCC)
    - type Job, Reservation, Service
    - user name
    - submitting `PID`
    - date of submission
    - date of completion (initially *null*)
    - description (text supplied by user)
  - scheduling information
    - scheduling class
    - scheduling priority
    - scheduling max shares
    - scheduling min shares
    - scheduling threads per share
    - scheduling memory size
    - scheduling memory units
  - job driver information
    - java command
    - java classpath
    - environment variables
    - user log directory
    - MQ broker
    - MQ queue
    - *Collection Reader* descriptor
    - *Collection Reader* overrides
    - getMeta timeout value
    - work item processing timeout value
    - work item processing exception handler
    - node identity
    - Linux Control Group limits
    - state
  - job process information (one or more instances)

    - java command
    - java classpath
    - environment variables
    - user log directory
    - MQ broker
    - MQ queue
    - deployment descriptor or aggregate data
    - initialization failure limits
    - node identity
    - Linux Control Group limits
    - state
    - service dependencies
  - service information (one or more instances)
    - See job process information above.
  - managed reservation information TBD
  - unmanaged reservation information TBD

Checkpoint Supervisor

The Checkpoint Supervisor provides functions to save and restore state information to/from persistent storage. State is stored whenever a significant change occurs. State is restored at Orchestrator boot time.

Saving and restoration of state facilitates reasonable continuity of service between Orchestrator lifetimes.

State Supervisor

The State Supervisor receives and examines publications from other DUCC components, records and distributes pertinent information obtained or derived, and advances state machines appropriately.

Publications are received from these components:

- Job Driver(s)
- Resource Manager
- Services Manager
- Agent(s) Inventory

Information from these sources is recorded in the `OR-map`. Based on information derived from all sources, the Orchestrator advances the state machines of currently active entities (Jobs, Reservations, Services). Once the `Completed` state is reached, the entity is no longer active on the cluster.

Note that `OR-map` is, in-turn, published at regular intervals for use by the other DUCC singleton and distributed components. The `OR-map` is the ”final authority” on the state of each Job, Reservation and Service currently or formerly deployed. See 16.3.1 `Controller`.

State Accounting Supervisor

The State Accounting Supervisor manages finite state machine for Jobs, Services, and Reservations. It provides functions to:

Advance from the current state to a next valid state

Advance from the current state immediately to the `Completed` state

Linux Control Group Supervisor

The Linux Control Group Supervisor assigns a maximum size (in bytes) and a composite unique identity to each *DUCC-Share*. This information is published for use by Agents to enforce Linux Control Group limitations on storage used by the corresponding running entity (for example, UIMA pipeline).

Employing Linux Control Groups is analogous to defining virtual machines of a certain size such that exceeding limits causes only the offending process to suffer any performance penalties, while other co-located well-behaved processes run unaffected.

Host Supervisor

The Host Supervisor is responsible for obtaining sufficient resource for deploying the Job Drivers for all submitted Jobs. It interacts with the Resource Manager to allocate and de-allocate resources for this purpose. It assigns a *JD-Share* to each active Job.

A *JD-Share* is a Linux Control Group controlled *DUCC-Share* of sufficient size into which a Job Driver can be deployed. A *JD-Share* is usually significantly smaller than a normal *DUCC-Share*.

Logging / As-User

The Logging and As-User modules permit the Orchestrator to write logging data into a file contained in ”user-space”, meaning a file into a directory writable by the submitting user, during processing of the submitted entity (Job, Managed Reservation...).

The Logging module also facilitates the recording to persistent storage of noteworthy events occurring during the Orchestrator lifetime. Noteworthiness is configurable as various levels, such as `INFO`, `DEBUG` and `TRACE`.

Administrators

The Administrators module grants users defined in the `ducc.administrators` file special privileges, such a being able cancel to any user’s Job.

Maintenance

The maintenance thread wakes-up at regular intervals to perform the following tasks:

Health

> The Orchestrator automatically caps Jobs and Services that exceed initialization error thresholds, and cancels those that exceed processing error thresholds.

MQ Reaper

> The Orchestrator cleans-up unused JobDriver AMQ permanent queues for Jobs that have completed.

Publication Pruning

The Orchestrator regularly publishes state for all active entities (Jobs, Reservations, Services). It also publishes state for recently completed ones. Pruning removes from regular Orchestrator publication completed entities that have been competed past a time threshold, nominally one minute.

Node Accounting

This module keeps track of each node's state, up or down. Nodes that do not report for a time exceeding a threshold, typically a few minutes, are considered down. This information is used for Jobs whose Job Driver advanced to the `Completed` state, whereby corresponding Job Processes on nodes that are reported down are marked as stopped by the Orchestrator, as opposed to waiting (potentially forever) for the corresponding Agent to report. This prevents Jobs from becoming unnecessarily stuck in the completing state.

16.3.2 Resource Manager (RM, also known as the Scheduler)

There is one Resource Manager per DUCC cluster.

The duties of the Resource Manager are:

fairly allocate constrained resources amongst valid user requests over time.

The Resource Manager provides essential functionality for operation of the DUCC system. It is configurable and tunable. It is also plug-replaceable.

The Resource Manager both publishes and receive reports. The Resource Manager receives Orchestrator publications comprising Jobs, Reservations, and Services as well as Agent publications comprising inventory and metrics. The Resource Manager publication occurs at regular intervals, each representing at the time of its publication the desired allocation of resources.

The Resource Manager considers various factors to make assignments, including:

supply of available nodes;

memory size of each available node;

demand for resource in terms of memory size and class of service comprising Jobs, Reservations and Services;

the most recent previous assignments and desirability for continuity;

The Orchestrator is the primary consumer of the Resource Manager publication which it uses to bring the cluster into compliance with the allocation assignments.

The Resource Manager adheres to the `IScheduler` interface. Algorithms adhering to this interface are eligible for replacing the DUCC supplied one.

Job Manager Converter

The Job Manager Converter module receives Orchestrator publications and updates its internal state with new, changed, and removed map entries comprising Jobs, Reservations and Services.

Node Stability

The Node Status module evaluates the health of the nodes within the cluster for consideration during resource scheduling. Any node deemed unhealthy is removed from the collection of available resources until such time as it is once again deemed healthy.

Node Status

The Node Status module receives Agent publications and updates its internal state with new, changed, and removed node status entries.

Resource Manager

The Resource Manager performs the following:

receive resource availability reports from Agents;

receive resource need requests the Orchestrator;

employ a scheduling algorithm at discrete time intervals to:

> consider the resource supply;
>
> consider the most recent allocation set;
>
> consider new, changed and removed resource demands;
>
> assign a resource to a request;
>
> remove a resource from a request;
>
> publish current allocation set;

Scheduler

The Scheduler runs at discrete time intervals. It assembles information about available nodes in the cluster. Each node, based upon its memory size is partitioned into zero or more *DUCC-Shares*. Each request (Job, Reservation and Service) is assessed as to the number of *DUCC-Shares* required based upon user-specified memory size. In addition, each request is assessed with respect to the user-specified class-of-service.

The Scheduler considers the most recent previous allocations along with changes to supply and demand. It then produces a new allocation set which the Resource Manager publishes as directions to the Orchestrator.

16.3.3 Services Manager (SM)

There is one Services Manager per DUCC cluster.

The duties of the Services Manager are:

facilitate services definition and persistence;

monitor and control services as dependencies on-demand.

The Services Manager provides additional functionality for operation of the DUCC system. It is configurable and tunable.

Although not essential for the main purpose of the DUCC system, in practical terms for large systems the Services Manager is highly desirable for improved resource utilization. By using shared services, resources are more effectively employed and work items processing is completed sooner.

Some dimensions:

- long warm-up time

  When the service takes a long time to warm-up, the Job in progress may have to sit idle for a long time before the first work item can be processed until the service upon which it depends has initialized and is ready. If the service is already up and ready, this delay can be avoided and the Job experiences no service delay.

- large storage use

  When the service has a large memory footprint, it can be far more efficient to have multiple Jobs share the service rather than having separate copies for each.

- short processing time

  Related to large storage use, if the time to process a work item is relatively short, again it can be much more efficient to share the service amongst multiple Jobs.

- not used

  If a service has not been used for a relatively long time, it may be better to shut it down and reclaim the resources for use elsewhere, especially on a busy cluster.

Ping Driver

This runs the watchdog thread for custom service pingers. It ascertains the liveness and healthiness of each service known to DUCC.

Service Handler

Carries out Service Manager validated request operations.

Service Manager, API Handler

Receives and validates service request operations:

- register
- unregister
- start
- stop
- query
- modify

The Services Manager maintains a registry of services. The attributes of these services may include one of more of the following:

`permanent` A permanent service is one that is kept up so long as the DUCC system is running.

`on-demand` An on-demand service is one that is kept up only during the lifetime of one or more Jobs that declare a dependency on the service

`lingering` A lingering service is one that continues for some limited time beyond the lifetime of the last dependent Job in anticipation of another Job arrival in the near future.

`dynamic` A dynamic service is one that automatically expands and contracts in terms of number of instances to meet demand.

`registered` A registered service is one that is pre-defined, whose definition is kept persistently and whose lifecycle is managed by DUCC.

`custom` A custom (unregistered) service is one that is not pre-defined, whose definition is not kept persistently and whose lifecycle is not managed by DUCC.

The Services Manager keeps within its registry information of two types: `service` and `meta`.

Type `service` information includes the following attributes:

- classpath
- description
- environment
- jvm
- jvm args
- log directory
- deployment descriptor
- failures limit
- memory size
- scheduling class
- linger time
- pinger classpath
- pinger log
- pinger timeout
- service endpoint
- working directory

Type `meta` information includes the following attributes:

- autostart
- endpoint
- implementors
- instances (count)
- identifier (number)
- ping-active
- ping-only
- service-active
- service-class
- service-health
- service-state
- service-statistics
- service-type
- stopped
- user
- uuid
- work-instances (PID list)

16.3.4 Process Manager(PM)

There is one Process Manager per DUCC cluster.

The duties of the Process Manager are:

monitor and control processes supporting analytic pipelines distributed over a collection of agent-managed nodes.

The Process Manager provides essential functionality for operation of the DUCC system.

The Process Manager both publishes and receive reports. The Process Manager receives Orchestrator publications comprising Jobs, Reservations, and Services. The Process Manager distributes two publications at regular intervals. One is heartbeat information to notify the Orchestrator and WebServer that the Process Manager is alive. The other is compacted Agent-destined information regarding processes that need to be started, stopped or modified.

The main function of the Process Manager is to efficiently manage the distributed Agents each of which manages processes running locally on its own respective node (machine, computer). The Process Manager interprets the Orchestrator publications and redistributes only the essentials to the collection of distributed Agents who each independently act to bring the state of locally deployed processes into compliance.

16.3.5 Agent

There is one Agent per node (machine, computer) per DUCC cluster. The Agents collectively provide essential functionality for operation of the DUCC system.

The duties of the Agent are:

deploy, monitor and control one or more processes supporting analytic pipelines on one node; and

publish inventory and metrics.

At the direction of the Process Manager, each Agent is instructed to manage its assigned *DUCC-Shares* by means of Linux Control Groups, and by injecting into them local process elements comprising Job Drivers, Job Processes, Service Processes, and Managed Processes.

The Agent is subdivided into several responsibility areas:

- Core
- Config
- Deploy
- Event
- Exceptions
- Launcher
- Metrics Collectors
- Monitor
- Processors

Core

The Agent publishes information about the state of the node (machine, computer) it controls. It also receives publications which it interprets to control processes deployed thereon. It also monitors activity on the node (machine, computer) and ensures that only sanctioned processes are running.

The Agent is normally launched at DUCC system start-up time. However, Agents may be started/stopped independently over time.

DUCC is only able to deploy user submitted applications to a node (machine, computer) upon which there exists an active Agent.

Config

- Agent Configuration

  The Agentconfigures itself according to the `ducc.properties` file. Aspects include:

  - launcher.thread.pool.size
  - launcher.process.stop.timeout
  - rogue.process.exclusion.filter

    Processes in this list are exempt for rogue process detection and termination.

  - rogue.process.user.exclusion.filter

    Users in this list are exempt for rogue process detection and termination.

  The Agentpublishes reports at configurable intervals:

  - Node Inventory

    Node Inventory is a report on the Agent-managed processes on this node.

  - Node Metrics

    Node Metrics is a report on the Agent-observed metrics on this node.

Deploy

- Managed UIMA Service

  The module is the Agent-managed integration between UIMA-AS and the user supplied application code which is deployed thereto.

Event

- Event Listener

  The module handles publication events:

  - Process Start

    A notification from the Process Manager to start a user submitted process constrained to a Resource Manager allocated number of *DUCC-Shares*.

  - Process Stop

    A notification from the Process Manager to stop a user submitted process.

  - Process Modify

    A notification from the Process Manager to modify a user submitted process.

  - Process Purge

    A notification from the Process Manager to purge a user submitted process.

  - Job State

    A notification from the Process Manager comprising abbreviated state of the DUCC-managed collection of entities: Jobs, Reservations and Services.

Launcher

The modules comprising the Launcher package are tasked with starting user processes on the Agent-managed node (machine, computer). The modules are:

- CGroups Manager

  This module provides functionality to partition the Agent-managed node (machine, computer) into *DUCC-Shares*, each *DUCC-Share* with limits on one or more aspects, including but not limited to memory and swap space.

  The CGroups Manager essentially starts, maintains, and stops instant virtual machines in correspondence with Resource Manager allocated *DUCC-Shares* into which user submitted processes are launched.

- Command Executor

  This module is the base class that provides functionality to launch a user specified process within the Resource Manager allocated *DUCC-Share*.

- DUCC Command Executor

  This module launches a user specified process within the Resource Manager allocated *DUCC-Share*. The process may be constrained by a Linux Control Group and may be spawned as the submitting user.

- JVM Args Parser

  The JVM Args Parser module extracts user specified JVM arguments for use in building an Agent-launchable subprocess comprising the user specified executable code.

- Launcher

  The Managed Process module provides virtual Agent capability.

  This module comprises a method used to launch multiple Agents on the same physical machine. It allows for the scale up Agents on a single machine to simulate load. Each Agent instance assumes a given name and IP address.

- Managed Process

  The Managed Process module manages a state machine for each Agent-managed user process. The states comprise:

  - Starting
  - Initializing
  - Ready
  - Failed
  - Stopped

- Process Stream Consumer

  The Process Stream Consumer module captures and redirects user process output to a log file.

Metrics Collectors

The modules comprising the Metrics Collectors package observe, calculate or otherwise gather specific metrics. Metrics collected are relative to these main categories:

- Garbage Collection Statistics
- Node CPU, Node CPU Usage, Node CPU Utilization
- Node Load Average

- Node Memory Info
- Node Users
- Process CPU Usage
- Process Major Faults
- Process Resident Memory
- Process Swap Usage

Monitor

The modules comprising the Monitor package observe various states and trigger actions when specific events occur.

- Agent Monitor

  When the Agent detects problems with the network, broker, or ping functions it terminates all Agent deployed processes.

- Rogue Process Detector

  The Agent detects aliens processes, those not expected for running the OS or DUCC or user processes deployed by DUCC. According to policy, the Agent may take one or more actions:

  – log an *alien detected* event

  – send notification to subscribers of alien detection events

  – with root privilege, signal the alien process to terminate

Processors

The modules comprising the Processors package assemble information for consideration when carrying out the Agent duties as well as for publication to other interested DUCC daemons. Information collected are relative to these main categories:

- Linux Node Metrics
- Linux Process Metrics
- Node Inventory
- Node Metrics
- Process Lifecycle
- Process Metrics

16.3.6 JobDriver(JD)

There is one JobDriver per Job.

The duties of the JobDriver are:

fetch analytic pipeline work items in correspondence with the user specified degree of parallelness;

dispatch work items to distributed analytic pipelines;

gather and report on performance statistics and errors;

retry failed recoverable work items; and

guarantee that individual work items are not mistakenly simultaneously processed by more than one analytic pipeline.

The JobDriver comprises a container into which the user specified *Collection Reader* is deployed. The JobDriver interacts with the user specified *Collection Reader* to fetch *CASes* (or *WorkItems*) for processing by a corresponding pipeline.

The JobDriver is deployed into a Resource Manager allocated *JD-Share* managed by a DUCC Agent.

The JobDriver is subdivided into several responsibility areas:

- Core
- Client
- Config
- Event

Core

- Job Driver

  The JobDriver module is the main thread. It setups and executes the JobDriver runtime environment.

  - initialize

    The initialize method sets-up all the machinery to fetch and process *CASes*.

  - run

    The run method manages all the machinery to fetch and process *CASes*.

    - wait for eligibility

      Do not start queuing *WorkItems* until at least one *Job Process* has initialized.

    - initialize UIMA-AS client

      Create an instance and one thread client for each corresponding *Job Process*thread.

    - queue *WorkItems*

      While terminate conditions are absent, repeat the process of queuing one work item for each thread, then sleeping for an interval, then rechecking for termination and performing additional queuing.

- Job Driver Component

  This module initializes the JobDriverfunction, receives and evaluates Orchestrator job mapswith respect to continuance or termination of the JobDriver, and triggers publication of JobDriverstatus reports.

- Job Driver Terminate Exception

  This module provides a means to identify the exception and possible reason for same when the JobDriver abnormally terminates.

- Synchronized Stats

  This module provides a method for the JobDriver to accumulate statistics in a thread safe manner. Per *WorkItem* statistics are maintained and produced:

  - number of *WorkItems*
  - minimum time to complete a *WorkItem*
  - maximum time to complete a *WorkItem*
  - average time to complete a *WorkItem*
  - standard deviation of time to complete a *WorkItem*

Client

- Callback State

  This module tracks *WorkItem* queuing state.

  Possible states are:

  - *PendingQueued*
  - *PendingAssigned*
  - *NotPending*

- *CAS* Dispatch Map

  This module tracks *WorkItems*. It comprises a map of *WorkItems* which includes node and Linux process identity.

- *CAS* Limbo

  Manage incomplete *WorkItems*. This module ensures that *WorkItems* are not simultaneously processed by multiple UIMA pipelines. It does not release *WorkItems* for retry processing elsewhere until confirmation is received that the previous attempt has been terminated.

- *CAS* Source

  Manage *CASes*. Employ the user provided CR to fetch *CASes* as needed to keep the available UIMA pipelines full until all *CASes* have been processed. Save and restore *CASes* that were pre-empted during periods of JP contraction, for example.

- *CAS* Tuple

  Manage *CAS* instance with meta-data.

  - UIMA *CAS* object.
  - DUCC assigned sequence number.
  - *CAS* retry status.
  - Job identifier.

- Client Thread Factory

  Produce one UIMA-ASclient thread instance for each *CAS* in-progress.

- Dymanic Thread Pool Executor

  Maintain a client-size thread pool for processing *CASes*. Each thread in the pool is assigned and tracks one *CAS* sent out for processing via *UIMA-AS sendAndReceiveCAS*. Each thread in the pool is re-usable once processing for the associated *CAS* is completed. The thread pool expands and contracts in correlation with the number of Resource Manager assigned *DUCC-Shares*.

- *WorkItem*

  The *WorkItem* represents one *CAS* to be processed, normally by one of the distributed UIMApipelines.

  - Manage and track the lifecycle of a *WorkItem* steps:
    - start
    - getCas
    - *UIMA-AS sendAndReceiveCAS*
    - ended or exception

- *WorkItem*

  Create a new *WorkItem* for given CasTuple. Associate a *WorkItem* with a UIMA-AS client thread.

- *WorkItem* Listener
- - onBeforeMessageSend

    Process callback that indicates work item has been placed on MQ queue and is awaiting grab by a JP.

  - onBeforeProcessCAS

    Process callback that indicates work item has been grabbed from MQ queue and is active in a UIMA pipeline. The associated node and Linux process identity are provided.

Config

The JobDriver publishes reports at configurable intervals:

- Job Driver Status Report

  Job Driver Status Report is a report on the JobDriver-managed *Collection Reader* sourced *CASes* (or *WorkItems*).

  Information includes *WorkItems* total-to-process, number-finished, number-failed, number-retried and other status.

Event

The module receives and handles publication events:

- `OR-map`

  The Orchestrator notification comprising the `OR-map` is the ”final authority” on the state of each Job, Reservation and Service currently or formerly deployed to the DUCC-managed cluster.

16.3.7 User Interface (UI)

There is one User Interface per DUCC cluster.

The duties of the User Interface are:

permit authorized users to submit, cancel and monitor distributed analytics; and

permit authorized users to administer the configuration and runtime aspects of the system.

The User Interface provides essential functionality for operation of the DUCC system. It comprises:

- Application Program Interface (API)
- Command Line Interface (CLI)

The User Interface facilities user and administrator operation of the DUCCsystem. Supported operations in create, retrieve, update, and modify.

The User Interface is subdivided into several responsibility areas:

- API
- CLI
- AIO
- WS

- JSON

API

Provide application program interface to the DUCC system comprising the ability to manage Jobs, Reservations and Services. See 16.3.7 CLI.

CLI

Provide command line interface to the DUCC system comprising the ability to manage Jobs, Reservations and Services.

- Submit Job
- Submit Reservation
- Submit Service
- Cancel Job
- Cancel Reservation
- Cancel Service
- Modify Job
- Modify Reservation
- Modify Service
- Query Job
- Query Reservation
- Query Service

AIO

Provide test and debugging support in preparation for distributed deployment to the DUCC system.

- All-In-One

  Create a process comprising a user submitted Job.

  Install the equivalent of a JobDriver.

  Install the equivalent of a *Job Process*.

  Process all work items.

- All-In-One Launcher

  Launch an All-In-One process locally or remotely, according to user specification.

- CAS Generator

  Employ the user specified *Collection Reader* to produce *WorkItems*.

- CAS Pipeline

  Employ the user specified *Analysis Engine* to process *WorkItems*.

- DD PArser

  Parse the user specified Deployment Descriptor.

Extract the *import* name.

- Message Handler

  User replaceable message handlers for info, debug, error, trace, etc.

WS

- Query Machines

  Fetch Machine facts.

- Query Reservation Facts

  Fetch Reservation facts.

- Query

  Fetch node facts.

JSON

- Machine Facts

  Provide information management with respect to each node (machine, computer) in the DUCC cluster, comprising status, ip, name, reserve, memory, swap, aliens, sharesTotal, sharesInuse, and heartbeat.

- Reservation Facts

  Provide information management with respect to each Reservation in the DUCC cluster, comprising id, start, end, user, rclass, state, reason, allocation, userProcesses, size, list, and description.

- Node Facts

  Provide information management with respect to each node (machine, computer) in the DUCC cluster, comprising a list of nodes and corresponding PIDs.

16.3.8 WebServer (WS)

There is one WebServer per DUCC cluster.

The duties of the WebServer are:

facilitate use of the Command Line Interface;

facilitate use of the Application Program Interface; and

facilitate use of additional complimentary utilities.

The WebServer provides complimentary functionality for operation of the DUCC system. It comprises:

monitor publications and files produced by:

- the OR
- the RM
- the SM
- the PM
- each Agent

provide user and administrator web pages to:

- permit authorized users to submit, cancel and monitor distributed analytics;

  - login and logout
  - monitor and control Jobs
  - monitor and control Services
  - monitor and control Reservations
  - monitor and control DUCC Administration
  - monitor and control DUCC Classes
  - monitor and control DUCC Daemons
  - monitor and control DUCC nodes (machines, computers)
  - display help
  - display manuals
  - control preferences
  - perform queries and filter results

provide runtime functionality to:

  - automatically cancel Jobs, Reservationsand Servicesbased upon client inactivity;
  - manage user authentication, sessions, and cookies
  - provide user customizable views
  - provide one-click access to deployed JVMs via jConsole

The WebServer is subdivided into several responsibility areas:

- ws
- config
- event
- jConsole
- registry
- server
- types
- utils
- root

ws

- Boot

  Initialize cache of Jobs, Reservation and Services from checkpoint and historical data repositories.

- Daemons Data

  Track DUCC daemons: status, name, boot time, host name, host ip, PID, publication size and max, heartbeat last and max, and JConsole URL.

- Ducc Data

  Track Jobs, Reservation and Services from Orchestrator publications.

- Machines Data

Track machines: status, machine name, machine IP, reserve size, memory size, alied PIDs, DUCC shares total and inuse, heartbeat last

config

event

jConsole

JConsoleWrapper provides the facility to one-click on a DUCC WebServerpage and be taken to the JConsole for the corresponding process.

registry

Manage user and administrator access to the Services Registry, comprising service and meta data.

server

- As User Employ *setuid* program to act on behalf of the user as the user, such as for writing log files.
- Cookies
- Handler
- Json Format
- Classic Format
- Web Monitor
- Web Properties
- Web Server
- Web Sessions Manage user sessions with the WebServer allowing privileged actions in the role of a user or an administrator. Provide login and logout facilities.

types

util

root

16.4 Interfaces

Interfaces description.

Chapter 17

Runtime

17.1 State Machines

17.1.1 Job State Machine

Table 17.1: Job State Machine

Id	Name	Next	Description
1	Received	2, 7	Job has been vetted, persisted, and assigned unique Id
2	WaitingForDriver	3, 4, 7	Process Manager is launching Job Driver
3	WaitingForServices	4, 7	Service Manager is checking/starting service dependencies for Job
4	WaitingForResources	5, 7	Scheduler is assigning resources to Job
5	Initializing	6, 7	Process Agents are initializing pipelines
6	Running	7	At least one Process Agent has reported process initialization complete
7	Completing	8	Job processing is completing
8	Completed		Job processing is completed

17.1.2 Service State Machine

Table 17.2: Service State Machine

Id	Name	Next	Description
1	Received	2, 3, 6	Service has been vetted, persisted, and assigned unique Id
2	WaitingForServices	3, 6	Service Manager is checking/starting service dependencies for Service
3	WaitingForResources	4, 6	Scheduler is assigning resources to Service
4	Initializing	5, 6	Process Agents are initializing pipelines
5	Running	6	At least one Process Agent has reported process initialization complete
6	Completing	7	Service processing is completing
7	Completed		Service processing is completed

17.1.3 Reservation State Machines

Table 17.3: Unmanaged Reservation State Machine

Id	Name	Next	Description
1	Received	2, 3	Reservation has been vetted, persisted, and assigned unique Id
2	Assigned	3	*DUCC-Shares* are assigned
3	Completed		*DUCC-Shares* not assigned

Table 17.4: Managed Reservation State Machine

Id	Name	Next	Description
1	Received	2, 3, 5	Reservation has been vetted, persisted, and assigned unique Id
2	WaitingForServices	3, 5	Service Manager is checking/starting service dependencies for Reservation
3	WaitingForResources	4, 5	Scheduler is assigning resources to Reservation
4	Running	5	Process Agent has reported program launched
5	Completing	6	Reservation processing is completing
6	Completed		Reservation processing is completed

17.2 Dependencies

17.3 Scheduling

17.4 Monitoring and Control

17.4.1 Automatic

17.4.2 Manual

17.5 Logging

17.5.1 System

17.5.2 User

17.6 Recovery

17.6.1 System

17.6.2 User