Distributed UIMA Cluster Computing

Ducc Team

Table of Contents

I. Introduction to DUCC ... 1
- 1. DUCC Overview ... 3
  - 1.1. What is DUCC? ... 3
  - 1.2. DUCC Job Model ... 3
  - 1.3. Default Collection Readers and CAS Consumers ... 5
  - 1.4. Error Management ... 5
  - 1.5. Cluster and Job Management ... 6
  - 1.6. Service Management ... 8
- 2. DUCC Application Quick Start ... 11
  - 2.1. Section 1 ... 11
- 3. DUCC Terminology, Acronuyms, and Glosssary ... 13
  - 3.1. Terms ... 13
  - 3.2. Acronyms ... 15

II. DUCC User's Guide ... 17
- 4. Command Line Interface (CLI) ... 19
  - 4.1. ducc_submit ... 20
  - 4.2. ducc_cancel ... 27
  - 4.3. ducc_reserve ... 27
  - 4.4. ducc_unreserve ... 28
  - 4.5. ducc_monitor ... 29
  - 4.6. ducc_service_submit ... 30
  - 4.7. ducc_service_cancel ... 34
  - 4.8. ducc_services ... 34
    - 4.8.1. ducc_service --register ... 36
    - 4.8.2. ducc_services --start ... 39
    - 4.8.3. ducc_services --stop ... 40
    - 4.8.4. ducc_services --modify ... 41
    - 4.8.5. ducc_services --query ... 43
    - 4.8.6. ducc_services --submit and --cancel ... 44
- 5. Job Logs ... 47
- 6. Application Programming Interface (API) ... 51
- 7. Webserver ... 53
  - 7.1. Common Links ... 53
  - 7.2. Jobs Page ... 53
  - 7.3. Job Details Page ... 55
  - 7.4. Reservation Details Page ... 57
- 8. Examples: Building and Testing a Simple Application ... 59

III. DUCC Administration Guide ... 61
- 9. Installation, Configuration, and Verification ... 63
  - 9.1. General Considerations ... 63
  - 9.2. Hardware Requirements ... 63
  - 9.3. Software Requirements ... 63
  - 9.4. Quick Installation Checklist ... 64
  - 9.5. Detailed Installation Procedures ... 65
    - 9.5.1. Basic System Initialization ... 65
    - 9.5.2. Install DUCC Distribution ... 65
    - 9.5.3. Perform Post-Installation Tasks ... 66
    - 9.5.4. Update ducc.properties ... 69
    - 9.5.5. Create the DUCC Node list ... 70
    - 9.5.6. Define the Job Driver nodepool ... 70
    - 9.5.7. Define the system administrators ... 71

9.6. Run The Verification Script 71
9.7. Start DUCC 71
9.8. Start DUCC Browser 73
9.9. Run a Job 73
9.10. Shutdown DUCC 74
10. Administration 77
10.1. ducc.properties 77
10.1.1. General DUCC Properties 78
10.1.2. Web Server Properties 84
10.1.3. Job Driver Properties 85
10.1.4. Service Manager Properties 88
10.1.5. Orchestrator Properties 89
10.1.6. Resource Manager Properties 92
10.1.7. Agent Properties 97
10.1.8. Process Manager Properties 102
10.1.9. Job Process Properties 104
10.2. ducc.classes 105
10.3. ducc.nodes 108
10.4. Nodepool Configuration 109
10.5. start_ducc 110
10.6. stop_ducc 112
10.7. check_ducc 113
10.8. verify_ducc 114
10.9. Logs 115
11. Resource Management, Operation, and Configuration 117
11.1. Overview 117
11.2. Scheduling policies 118
11.3. Priority vs Weight 119
11.4. Node Pools 120
11.5. Job Classes 120

Part I. Introduction to DUCC

Chapter 1. DUCC Overview

The source for this chapter is ducc_ducbook/documents/part-introduction/chapter-overview.xml.

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 DUCC job model is defined in terms of the UIMA and UIMA-AS framework. A 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, “Standard UIMA Pipeline” [3].

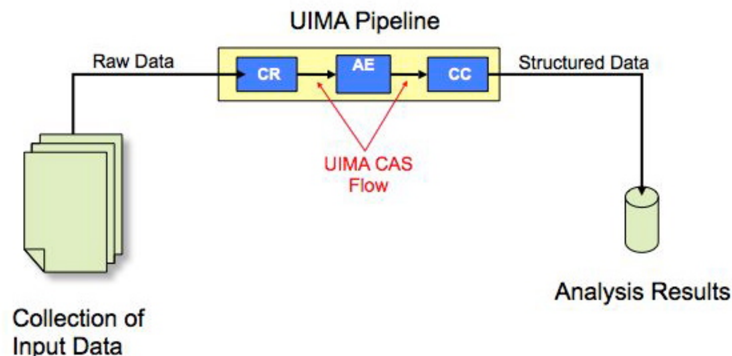

Figure 1.1. Standard UIMA Pipeline

With UIMA-AS the CR is separated into a discrete process and a CAS Multiplier is introduced into the analytic pipeline as an interface between the CR and the pipeline, as shown in Figure 1.2, “UIMA Pipeline As Scaled by UIMA-AS” [4]. Multiple analytic pipelines are serviced by the CR and are scaled-out over a computing cluster.

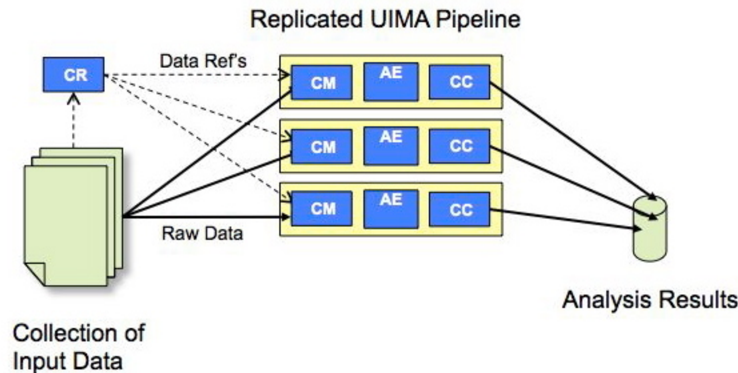

Figure 1.2. UIMA Pipeline As Scaled by UIMA-AS

Under DUCC, the Collection Reader is executed in a process called the Job Driver (or JD). The analytic 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 and to direct CASs to the JPs. Similarly the JP provides a thin wrapper over the analytics as shown in Figure 1.3, "UIMA Pipeline As Automatically Scaled Out By DUCC" [4].

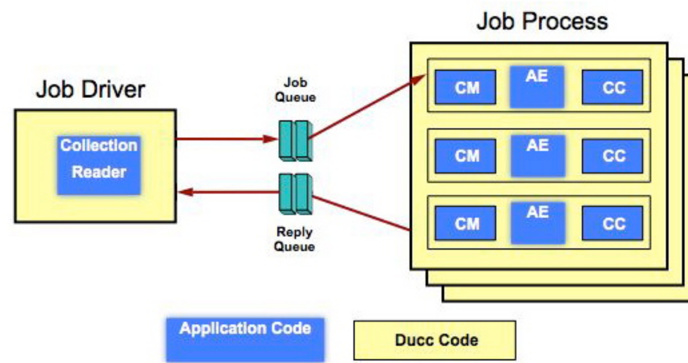

Figure 1.3. UIMA Pipeline As Automatically Scaled Out By DUCC

On job submission, the DUCC CLI inspects the XML defining the analytic and generates a UIMA-AS Deployment Descriptor (DD) from it. DUCC generates job-unique queue endpoints, setups up the queues, and sets up multiple pipeline threads so that the entire transformation from the user's core-UIMA job to full UIMA-AS scalout is transparent and automatic. (Users may supply their own CM but it is not necessary as DUCC provides a default CM.) A simple collection of parameters, known as the Job Specification (essentially a Java properties file) defines the CR, CM, AE, and CC, threading level, logging parameters, etc. Taken together the Job Descriptor, Job Driver, and set of Job Processes comprise a DUCC job.

Users may want to provide their own DDs to more fully control the pipeline in the JPs. This model is also support by DUCC; see Figure 1.4, "UIMA Pipeline With User-Supplied DD as Automatically Scaled Out By DUCC" [5].

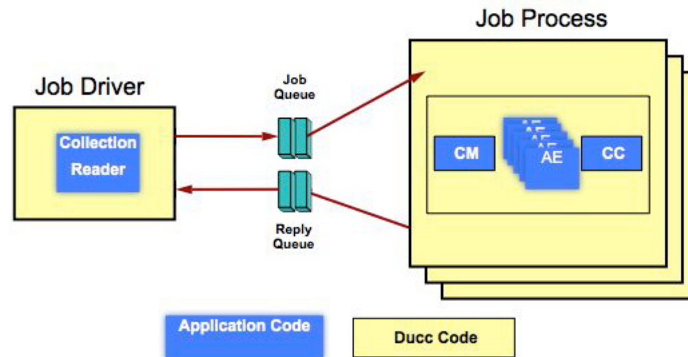

Figure 1.4. UIMA Pipeline With User-Supplied DD as Automatically Scaled Out By DUCC

The DUCC Job Descriptor includes properties to enable automated management and scale-out over large computing clusters. Such management includes multiple-user support (jobs run under the identity of the submitting user), a fair-share scheduler capable of balancing resources among all users, automated performance monitoring via the UIMA-AS monitoring facilities, display of job status and performance statistics via a built-in web server, and error-handling of the UIMA pipelines, also using the UIMA-AS facilities.

DUCC provides a Command Line Interface (CLI) to submit UIMA pipelines for execution as jobs. (An Application Programming Interface (API) is in progress but not available with the current releaase.) The CLI inspects the pipeline XML descriptors (as named in the Job Specification) and automatically generates UIMA-AS Deployment Descriptors. The descriptors are passed to the DUCC orchestration tools which establish the Collection Reader inside a Job Driver (JD) process as a UIMA-AS service client. The Job Specification is given to the Resource Manager which returns the identities of the nodes where the JPs (Job Processes) are to be run. Finally the JPs are started with the pipeline's AEs as UIMA-AS services and the JD starts the CRs which begin delivering CASs. Endpoint management, creation of the DD, spawning and management of the CR and AEs are all automated by DUCC.

1.3. Default Collection Readers and CAS Consumers

Descibe what we provide - the zip CR and zip CM or equivalent, with some motherhood about why using these is good, but pointing out that users are free to make their own.

What will these be - the moral equivalent of the zip reader and the one used for NLP?

1.4. Error Management

A classic problem of large distributed systems is error management. Small errors can scale-out so that a single typo or oversight can flood the system with redundant error notifications and waste significant resources with useless computation. It can also be very difficult to isolate errors which can occur anywhere in the network. To manage this process DUCC provides a number of features.

DUCC uses the UIMA-AS error-handling facilities to reflect errors from the JPs to the JDs. The JD wrappers implement logic to enforce error thresholds, to log the errors coherently, and to inform the web server. All error thresholds are configurable. Additionally, the user may implement custom

logic to determine whether errors should be considered fatal or transient, and the number of failures to tolerate. Each job may provide its own fully customized error handling policy.

A large UIMA application can take significant time initializing, reading from databases, and so on. The initialization process itself can be fragile and error-prone. It would be wasteful and useless to allow such an application to be scheduled on a large number of cluster nodes only to fail. To manage this, DUCC enforces two policies:

- JPs are allowed a maximum number of failures in the initialization stage before DUCC terminates the job.
- A minimum number of processes is allocated to a job when it starts. The job is not allocated additional processes until at least one JP completes the initialization phase, at which point the job becomes eligible for more processes.

Once a JP is initialized the error handling is slightly different. Errors at this stage may be transient: network failures, service failures, etc. Or they may be systemic in the application (bugs). DUCC allows a maximum number of JP failures after initialization and if the threshold is exceeded, the job is terminated. If a process has a failure, the failure is reflected back to the JP and the process is terminated. If the threshold has not yet been exceeded, the Resource Manager will allocate space and a new process will be started.

In all failure scenarios, DUCC attempts to capture the associated stack traces and error messages and presents them as links in the job pages from the DUCC web server.

1.5. Cluster and Job Management

Distributing work over multiple physical processors on a network can be difficult to manage, even for relatively small numbers of processors. DUCC provides extensive tooling manage the cluster and the jobs running on it.

Multiple User Support. DUCC runs all work under the identity of the submitting user. This provides a level of security and privacy for each user and their job. Logs are written with the user's credentials into the user's file space designated at job submission, enabling users to manage them as needed.

Fair-Share Scheduling. DUCC is intended to support UIMA processing of natural language. This work is inherently much more memory intensive than it is CPU intensive. In order to insure that the pipelines execute efficiently, nodes need to be allocated according to the amount of real RAM they support. With few exceptions, these jobs encounter bottlenecks on real memory well before CPU bottlenecks.

To manage this, DUCC contains a scheduler designed to allocate nodes in the cluster according to declared memory usage. All RAM is treated as a single, distributed pool of memory. "Fair" share means that the memory is allocated such that each user is allocated the same amount regardless of the number of jobs the user has submitted. Each user's fair-share is then divided equally among all their jobs. Machines are then allocated to jobs so the total memory in the machines assigned to a user is the same as their fair-share. Often some users don't need (and can't use) their fair share, in which case the DUCC scheduler allocates the lefovers to users that are able to use it.

The DUCC scheduler provides the ability to "weight" some users so they are allocated more than their simple fair-share of memory. There is also a priority scheme that insures some types of work are always scheduled irregardless of fair-share considerations. There is also a mechanism for partitioning the nodes according to arbitrary constraints ("closeness" to constrained resources,

priority usage such as "production" vs. "development" use, etc.), and assigning jobs to specific partitions or "nodepools".

DUCC assumes that most jobs written to the UIMA framework are fully parallel and that individual processes can be evicted as needed; as well it assumes that process can be added to a job if resources are available. The DUCC scheduler uses these properties to dynamically expand or reduce the number of processes assigned to jobs, according to fair-share policies and the amount of work in the system. For example, if a new user submits a job, this will generally reduce everybody's fair-share, and result in some processes being evicted to make room for the new user. Similarly, if all of a user's jobs exit, then the remaining jobs will be allocated the resources that are now freed. Note that if a user who already has jobs running submits a new job, then *only* that user's jobs are affected because his/her fair-share is the same, and has to accomodate new work.

Some jobs may not be parallel, or for some other reason, cannot tolerate being evicted and restarted. The DUCC scheduler implements a policy to allow jobs with "fixed" (or "pinned") node allocations whic prevents those jobs from being preempted; conversly the "fixed" policy prevents those jobs from growing. Thus, once scheduled, this type of job is "fixed" in place and will never move to different nodes.

The scheduler also supports the concept of *Reservations*. A reservation has no job associated with it; users are allowed to use the reserved resources as they wish (within reason). Reservations for full (dedicated) nodes or for partial nodes (based on RAM) are supported.

Job Lifetime Management and Orchestration. DUCC includes an *orchestrator* that manages the lifetimes of all jobs, services, and reservations. Jobs are submitted to the orchestrator, which is responsible for insuring pre-requisite services are available and that resources are scheduled for the job. It starts the job's JD and JP processes and signals the JD to start servicing work to the JPs. It is also responsible to keep the scheduler and web server apprised of the status of all jobs.

DUCC Agents. A process called the DUCC *Agent* is run on each node managed by DUCC. This process has several roles:

- Manage JP and JD processes. The agent starts, stop, and manages the life cycle of these processes. It also monitors performance statistics on behalf of these processes, reporting to the web server.
- Monitor node performance and "aliveness". The agent monitors CPU, memory, etc, and provides the information in regular *heartbeats* that are watched by the Resource Manager and Web server for scheduling and reporting purposes.
- Watch for *rogue processes*. The agents watch for processes not associated with DUCC jobs or other DUCC-initiated work and reports to the web server. Administrators are then able to easily identify and reap processes that may be interfering with DUCC jobs.

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

- All jobs in the system with relevant information: user, times, work finished work completed, processes allocated, and many other. For each job, additional pages provide details including node, PID, stat status of all work items, the submitted job specification, etc. If errors occur, links from the job entry to the errors in the logs are provided.
- All reserved nodes with relevant information: user, times, nodes, processes running in the reservation, etc.
- All nodes in the system and their status, usage, etc.

- All services and rel-event information: user, nodes, usage, who is using the services, queue size, etc.
- The status of all DUCC management processes.

Management Scripting. DUCC provides rich scripting support to:

- Start and stop full DUCC systems.
- Start and stop and 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.

1.6. Service Management

Overview. *Services*, in the context of DUCC, are long-running processes that await requests from UIMA pipeline components and return something in response. Services can be any arbitrary process using any arbitrary communication protocol but in the current version of DUCC only UIMA-AS services are fully supported.

The DUCC service manager implements several high-level functions:

- Insure services are available for jobs before allowing the jobs to start. This fail-fast prevents unncessary 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 amis.
- Automate the startup, care, and management of services.
- Report on the state of services: processes, queue depths, comsumers, and so on.

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.
- CUSTOM. This is any arbitrary service. DUCC supports monitoring of CUSTOM services and performs job dependency checks, but (in the current version) does not support start and stop of CUSTOM services.

Service Endpoints. Services are referenced by a specifier called a *service endpoint.*. The service endpoint is a formatted string indicating:

- The service type: UIMA-AS or CUSTOM.
- The service name. For UIMA-AS services, this is the name of the queue in the ActiveMq Broker used for communication with the service. For CUSTOM services this is any arbitrary string as dictated by the service. Service names must be unique within the system.
- For UIMA-AS services only, the URL of the ActiveMq broker.

Dependent and Pre-Requisite Services and Jobs. A *dependent service* is a service which is dependent on at least one service to perform it's function. A *dependent job* is a job which is dependent on at least one service to perform it's function.

A *pre-requisite service* is a service which is required by another job or service. (Note that there are no pre-requisite jobs.)

Service Classes. Services may be started externally to DUCC, explicitly through DUCC as a job, or as registered services. These form three natural classes of services with slightly different management characteristics.

Implicit Services. An *implicit service* is started externally to DUCC and discovered by DUCC only when it is referenced by a job's *service-dependency* parameter. On submission of a job with a dependency on an implicit service, the SM sets up a "ping" thread that check if the service exists at the endpoint. If so, the SM adds the service to its list of known services and marks the job "ready to schedule". If the service is a UIMA-AS service the SM establishes a monitor thread on the queue for reporting purposes. The service is monitored throughout the lifetime of the job. If the service should stop responding, its state is updated as "not-responding" but the job is allowed to continue as DUCC cannot tell if the job is still using it or not, or if the outage is temporary. If the job is a CUSTOM service, the service owner may specifiy custom code to run in the ping thread; for CUSTOM services, this same code is used to run both ping and monitor functions.

When the job exits, a timer is set and DUCC continues to monitor the service against the possibility that subsequent jobs will need it. Once the last job using the service has exited and the service timer expired, the SM stops the monitors and purges the service from its records.

Submitted Services. A *submitted service* is a service that is submitted to DUCC as a job. A submitted service is essentially a normal DD-style job (a job in which the user supplies the full UIMA-AS DD), but without a Collection Reader. Because DUCC is managing this service it can provide more support than for *implicit services.*

Submitted services can be dependent upon other services. When such a service enters the system, DUCC verifies it's pre-requisite services. When (or if) all pre-requisite services are availble DUCC marks the new service "ready to schedule". The lifecycle of the service is monitored so that dependent services and jobs are marked "ready to schedule" only after the submitted service has completed its initialization phase. A ping thread and queue monitor are also started against the newly submitted service. If the submitted service is unable to successfully initialize, services and jobs that are dependent on it are marked "not runnable" and the DUCC Orchestrator cancels them.

DUCC manages the lifecycle of submitted services, but because they are submitted by entities other than DUCC, the SM performs no additional management for them. When a submitted service is canceled by its owner, DUCC stops the ping and queue monitors. Any jobs or services dependent on it are allowed to continue until they complete or fail due to unavailability of the service.

Registered Services. *Registered services* are fully managed by DUCC. A service is registered with DUCC using the CLI to provide the full job specification of the service, the initial number of instances of the service, and whether the service should be automatically started when DUCC itself is started. Registered services started when DUCC is started are called *automatic* services. Registered services that are started only when referenced by other dependent jobs or services are called *on-demand* services. The service is registered with the submitter's credentials and is run with that user's credentials when it is started.

Automatic Services. An *automatic* service is a registered service that is flagged to be 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 startup. The SM submits the registered service specification on behalf of its owner. Each such submission is for a single service instance. If found, the SM repeatedly submits the specification until the registered number of instances is reached.

Ping and monitor threads are started. Jobs and other services may use these services in the same manner as submitted services. If an automatic service instance should die or be canceled out of the scope of the SM, the SM will restart the instance, maintaining the registered number of instances at all time. *Automatic* services are not terminated when their dependent jobs/services exit; they're termanted only when DUCC itself is terminated, or by use of the service *stop* command.

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. f the service is already started, the dependent job/service is marked ready to schedule as indicated above. If not, the service registry is checked and if a start-on-demand service with an endpoint matching the *service-dependency* is found, DUCC submits the service on behalf of the service owner (in the same manner as for automatic servic establishing the registered number of service instances, a ping thread, and a monitor). When the service has completed initialization the dependent job/service is marked ready to schedule. If the on-demand service cannot be found in the registery, the referring entity is marked not-startable and the DUCC Orchestrator cancels it.

Subsequent jobs and services that reference the on-demand service will use the started instances. When the last job/service that references the on-demand service exits, a (configurable) 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 exipires, and there are no more dependent jobs/services, the on-demand service is automatically stopped to free up its resources for other work.

Registered Service Management. The CLI for registered services provides several functions:

Register
Register files a service specification with the SM. The service may optionally be started as part of registration. The service definition and state is persisted over system restarts and is deleted only with the Unregister function.

Unregister
Unregister removes the service specification. The service is stopped if it is started and not busy. (Note that if the service is busy, jobs and services that are dependent on it may subsequently fail.)

Modify
Modify allows dynamic update of some parameters of registered services:

- *Automatic* and *On-Demand* state.
- The minimum number of service instances to start when the service is started.

Start
Start submits the service specification to the DUCC Orchestrator (repeatedly, until the correct number of instances are started). If the service is explicitly started with the **start** CLI, the service continues to run even after the last reference is gone, regardless of whether it is automatic or on-demand. Start is also used to increase the number of running instances of a service. The registry may be optionally updated to reflect the new number of started instances.

Stop
Stop stops the instances for a registered service. The registry may be optionally updated to reflect the new number of instances that are still running.

Query
A CLI-based query is supplied to report on all services known to DUCC, their states, their instances, their dependent jobs, and performance statistics for the service.

Chapter 2. DUCC Application Quick Start

The source for this chapter is ducc_ducbook/documents/introduction/quick-start.xml

2.1. Section 1

This Sentence Intentionally Left Blank

Chapter 3. DUCC Terminology, Acronuyms, and Glosssary

The source for this chapter is ducc_ducbook/documents/introduction/terminology.xml

3.1. Terms

This section defines terms and phrases as used in the context of DUCC.

Automatic Service
: An *automatic 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 job or service that specifies one or more service endpoint in their job specification. The service or job is dependent upon the referenced service being operational before being started by DUCC.

DUCC
: DUCC stands for "Distributed UIMA Cluster Computing."

Implicit service
: An *emplicit service* is a service that is started externally to DUCC but referenced by some dependent service or job.

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. DUCC manages the usage of the service and (in a future verseion of DUCC) automatically increases and decreases the number of service instances as dictated by demand.

On-Demand Service
: An *on-demand service* is a registered service that is not started when DUCC is started. Instead, the service is started when referenced in some job or services service dependency, and stopped when the referencing entity exits.

Service Instance
: A *service instance* is one physical process which runs a *CUSTOM* or *UIMA-AS* service.

Orchestrator (OR)
: The Orchestrator coordinates all work in the system. All new work enters through the orchestrator which guides it through the various DUCC components.

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

Resource Manager (RM)
: The Resource Manager allocates and schedules physical resources among the jobs.

Service Class
: The three *service classes* are

- *implicit*, referring to a service started independently from DUCC,
- *submitted*, referring to a service submitted as a job to DUCC, and
- *registered*, referring to a registered DUCC service.

Service Endpoint
In DUCC, the *service endpoint* provides a unique identifier for a service and in the case of UIMA-AS services, a well-known address for contacting the service. For CUSTOM services, the endpoint is of the form CUSTOM:*string* where *string* is any alphanumeric string provided by the service owner. For UIMA-AS services, the endpoint is of the form UIMA-AS:*queue name*:*ActiveMQ broker URL*.

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 also monitor nodes, sending heartbeat packets with node statistics to interested components (such as the RM and web-server). All Job Driver and Job Process processes are managed as children of the agents.

Ducc-mon
Ducc-mon is the DUCC web-server. All DUCC state of import or interest is presented here including job state, cluster state, DUCC daemon state, and visualization of the system. Various controlling actions such as canceling jobs, submitting reservations, and administrative functions are supported.

Job Driver (JD)
The Job Driver is a thin Java 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 java wrapper that encapsulates a job's Analysis Engine. 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 a job. It identifies the UIMA components (CR, AE, etc) that comprise the job, and it specifies system-wide properties of the job (classpaths, RAM requirements, etc). The properties may be provided as (key, value) pairs to the CLI/API, or in a Java propeties file.

Job
A DUCC job consists of the components required to deploy and execute a UIMA pipeline over a computing cluster. It consist 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
In DUCC, a "share quantum" refers to some quantity of memory; for example, 15GB. The RM schedules resources according to share quanta. The share quantum is the smallest unit of memory that can be assigned. See the section describing the Resource Manager for details.

The terms "share" and "share quantum" are synonymous in DUCC.

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).

From the Resource Management view, a process is comprised of one or more share quanta.

Weighted Fair Share

The Weighted Fair Share calculation is used to apportion resources in a "fair" manner to the outstanding work in the system. To account for some work being more "important" than others, a weighting factor may be applied to bias the fair-share calculations in favor of such work.

See the Resource Manager section for more details on Weighted Fair Share in DUCC.

Work Items

A *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 CR to a UIMA service. It could be thought of as a single "question" to be answered by a UIMA analytic. Usually each DUCC JP executes many work items per job.

3.2. Acronyms

This section defines acronims as used in the context of DUCC.

AE: UIMA Analysis Engine

CAS: UIMA Common Analysis Structure

CC: CAS Consumer

CM: UIMA CAS Multiplier

CR: UIMA Collection Reader

DUCC: Distributed UIMA Cluster Computing

JD: Job Driver

JP: Job Process

OR: Orchestrator

PM: Process Manager

RM: Resource Manager

SM: Service Manager

UIMA: Unstructured Information Management Architecture (see http://uima.apache.org/)

UIMA-AS: UIMA Asynchronous Scaleout (see http://uima.apache.org/doc-uimaas-what.html)

Part II. DUCC User's Guide

Chapter 4. Command Line Interface (CLI)

The source for this chapter is ducc_ducbook/documents/part-user/chapter-cli.xml

The Command Line Interface is provided in several forms:

1. A Java "main" class, suitable for invoking from user-supplied scripting such as Ant or Python. Users of this must set the Java CLASSPATH to include a subset of the jar files supplied with DUCC.

   To run the commands directly from Java the CLASSPATH must be set correctly and an environment variable, `DUCC_HOME` must be set.

   DUCC_HOME
   Set DUCC_HOME to the location where DUCC is installed. For example:

   ```
   export DUCC_HOME=/home/ducc/ducc_runtime
   ```

   CLASSPATH
   The CLASSPATH must include all of the following elements, relative to DUCC_HOME:

   $DUCC_HOME/lib/ducc-cli.jar
   $DUCC_HOME/lib/ducc-common.jar
   $DUCC_HOME/lib/apache-activemq-5.5.0/activemq-all-5.5.0.jar
   $DUCC_HOME/lib/apache-commons-cli-1.2/commons-cli-1.2.jar
   $DUCC_HOME/lib/apache-camel-2.7.1/*
   $DUCC_HOME/lib/http-client/*
   $DUCC_HOME/lib/springframework-3.0.5/*
   $DUCC_HOME/uima/*
   $DUCC_HOME/resources

2. Executable jars for each CLI command. These obiviate the need to establish a classpath but do require DUCC_HOME to be set:

   ```
   export DUCC_HOME=/home/ducc/ducc_runtime
   ```

3. A script wrapper to the Java "main" that completely establishes the environment. These wrappers use the execubable jars, establishing the DUCC environment and obiviating the need to set DUCC_HOME.

   While not required, it may be useful to put the DUCC bin directory into your path:

   ```
   export PATH=$PATH:/home/ducc/ducc_runtime/bin
   ```

The following actions may be taken using the CLI:

1. Submit a job for ececution.
2. Cancel a job in progress.
3. Request a reservation of full or partial machines.
4. Cancel a reservation.

5. Monitor the progress of a job that is already submitted.
6. Submit a service for execution.
7. Cancel a service.
8. Register a service.
9. Unegister a service.
10. Start a registered service (if not auto-started).
11. Stop a registered service.

The next sections describe these actions in detail.

4.1. ducc_submit

The source for this section is ducc_ducbook/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

Executble Jar
java -jar *$DUCC_HOME*/lib/ducc-submit.jar

Java main
org.apache.uima.ducc.cli.DuccJobSubmit

If no options are given, help text is presented.

Options:

--cancel_job_on_interrupt

If the job is started with *--wait_for_completion*, this option causes the job to be canceled with Ctrl-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_completin* is not specified this option is ignored.

--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.

--driver_classpath [classpath]

This is the classpath for the Job Driver, necessary for DUCC to find the Collection Reader.

--driver_descriptor_CR [descriptor.xml]

This is the XML descriptor for the Collection Reader. It is searched for as a resource as described above.

--driver_descriptor_CR_overrides [list]

This is the Job Driver collection reader configuration overrides. They are specified as name/value pairs in a comma-delimeted list. For example:

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

--driver_environment

This specifies environment parameters for the Job Driver. If present, they are added to the Job Driver's environment as the process is spawned. It must be a quoted, blank-delimeted lsit of name-value pairs. For example:

```
"TERM=xterm DISPLAY=:1.0"
```

Note: On Secure Linux systems, the environemnt variable LD_LIBRARY_PATH may not be passed to the user's program. If it is necessary to pass LD_LIBRARY_PATH to the JP or JD processes, it must be specified as DUCC_LD_LIBRARY_PATH. Ducc (*securely*) passes this as LD_LIBRARY_PATH, *after* the JP or JD has assumed the user's identity. For example:

```
"--process_environment TERM=xterm DISPLAY=:1.0 DUCC_LD_LIBRARY_PATH=/my/own/
```

--driver_jvm_args

This specifes extra JVM arguments to be provided to the Job Driver process. It is a blank-delimeted list of strings. Example:

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

--driver_memory_size [size-in-GB]

This specifies the size of memory for the Job Driver, in GB. Example:

```
--driver_memory_size 16
```

.

--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. 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 the user's home directory. 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 is descripbed in Chapter 5, *Job Logs* [47].

> **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_classpath [ClASSPATH]

This specifies the Java CLASSPATH to use in each Job Process (JP) and must be specified. Example:

```
--process_classpath a.jar:b.jar
```

.

--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 CLASSPATH and data path as described in the *notes [26]*. For example:

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

--process_deployments_max [integer]

This specifies the maximum nunber 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 number of pending work items still to be done in the job.

```
--process_deployments_max 66
```

.

--process_descriptor_AE [descriptor]

This specifies Analysis Engine descriptor to be deployed in the Job Processes. This descriptor is a resource that is searched for in the CLASSPATH and data path as described in the *notes*. It is mutually exclusive with *--process_DD* For example:

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

--process_descriptor_AE_overrides [list]

This specifies AE overrides. It is a comma-delimeted 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 CLASSPATH and data path as described in the *notes*. 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-delimeted 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 CLASSPATH and data path as described in the *notes*. 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-delimeted list of name/value pairs. Example:

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

--process_environment [environment]

This specifies environment parameters for the Job Processes. If present, they are added to the Job Process environment as the process is spawned. It must be a quoted, blank-delimeted lsit of name-value pairs. For example:

```
"--process_environment TERM=xterm DISPLAY=:1.0"
```

Note: On Secure Linux systems, the environemnt variable LD_LIBRARY_PATH may not be passed to the user's program. If it is necessary to pass LD_LIBRARY_PATH to the JP or JD processes, it must be specified as DUCC_LD_LIBRARY_PATH. Ducc (*securely*) passes this as LD_LIBRARY_PATH, *after* the JP or JD has assumed the user's identity. For example:

```
"--process_environment TERM=xterm DISPLAY=:1.0 DUCC_LD_LIBRARY_PATH=/my/own/
```

--process_failures_limit [integer]

This specifies the maximum number of individual Job Process (JP) failures that are to be tolerated before killing the job. The default is 15. If this limit is exceeded over the lifetime of a job DUCC terminates the entire job.

```
"--process_failures_limit 23"
```

--process_get_meta_time_max [integer]

When a job is started the Job Driver issus a single "get-meta" requests to the (DUCC-generated) endpoint of the JP processes for the job to insure that at least one UIMA-AS server processes for the job have started. This parameter specifies the time in seconds to wait for a response. If the request times out the Job Driver assumes that no UIMA-AS service for the job was able to start and it terminates the job. If not specified, the timeout is 2 minutes. Example:

```
"--process_get_meta_time_max 10"
```

--process_initialization_failures_cap [integer]

This specifies the maximum number of independent Job Process initialization failures (i.e. System.exit(), kill-15...) before the number of Job Processes is capped at the number in state Running currently. The default is 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_jvm_args [list]

This specifies additinal arguments to be passed to the Job Process JVM. Example:

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

--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. if this amount is exceeded by a Job Process the Agent terminates the process with a ShareSizeExceeded message. Example:

```
--process_memory_size 33
```

--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 Agent to determine howmany 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 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. Example:

```
--schedling_class normal
```

--service_dependency[list]

This specifies a comma-delimeted list of services the job processes are dependent upon. Each endpoint must be of the form UIMA-AS:*endpoint*:*broker_url* where *endpoint* is the UIMA-AS service endpoint and *broker_url* is the ActiveMQ broker URL.

In the example are two dependencies, one with endpoint RandomSleepAE and broker tcp:bluej682:61616, and the other with endpoint OtherEp and broker URL tcp:bluej123:123. Example:

```
--service_dependency  UIMA-AS:RandomSleepAE:tcp:bluej682:61616, \
                UIMA-AS:OtherEp:tcp:bluej123:123
```

--specifiecaiton [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).

For example,

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

where the *job.props* contains:

```
working_directory=/Users/challngr/projects/ducc/ducc_test/test/bin
process_get_meta_time_max=5
process_failures_limit=20
driver_descriptor_CR=org.apache.uima.ducc.test.randomsleep.FixedSleepCR
driver_environment=DUCC_LD_LIBRARY_PATH=/a/other/bogus/path
process_environment=AE_INIT_TIME=10000 DUCC_LD_LIBRARY_PATH=/a/bogus/path
driver_classpath=/home/bob/duccapps/ducky_driver.jar
log_directory=/Users/challngr/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
process_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
```

--timestamp

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 does not return control to the consoke immediately, and instead monitors the DUCC state traffic and prints information about the job 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. Example:

```
--working_directory /Users/challngr/projects/ducc/ducc_test/bin
```

Notes:

When searching for UIMA XML resource files such as descriptors, DUCC searches both the classpath and the data path according to the following rules:

1. If the resource ends in .xml it is assumed the resource is a file and the path is either an absolute path or a path relative to the specified working directory. If the file is not found the search exits and the job is terminated.

2. If the resource does not end in .xml, DUCC creates a path by replacing the "." separators with "/" and appending ".xml". It then searches two places:

    a. The user's CLASSPATH as a file (that is, not in a jar), and

    b. In the jar files provided in the user's CLASSPATH.

    If the resource is found in either place the search is successful. Otherwise the search fails and the job is terminated.

The resource search-order rules apply to all of the following submit parameters:

- --driver_descriptor_CR
- --process_descriptor_AE
- --process_descriptor_CC
- --process_descriptor_CM

4.2. ducc_cancel

The source for this section is ducc_ducbook/documents/part-user/cli/cancel.xml

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

Executable Jar
java-jar *$DUCC_HOME*/lib/ducc_cancel.jar

Java main
org.apache.uima.ducc.cli.DuccJobCancel

If no options are given, help text is presented.

Options:

--id [jobid]

The ID is the jobid returned by the job submission.

--help

Prints the usage text to the console.

4.3. ducc_reserve

The source for this section is ducc_ducbook/documents/part-user/cli/reserve.xml

Notes:

Reservations may be for full machines, or partial machines based on memory. The mechanism for distinguishing which type of reservation 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 reservations.

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.

Usage:

Script wrapper
$DUCC_HOME/bin/ducc_reserve

Executble Jar
java -jar *$DUCC_HOME*/lib/ducc-reserve.jar

Java main
org.apache.uima.ducc.cli.DuccReservationSubmit

If no options are given, help text is presented.

Options:

--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.

--number-of-instances [integer]

This specifies the number of full or partial machine reservations to schedule.

--instance-memory-size [KB|MB|GB|TB]]

This specifies the amount of memory the reserved machine must supoprt. 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.

--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.

--specifiecaiton [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).

4.4. ducc_unreserve

The source for this section is ducc_ducbook/documents/part-user/cli/unreserve.xml

Description:

The *unreserve* CLI is used to release reserved resources.

Usage:

Script wrapper
$DUCC_HOME/bin/ducc_unreserve

Executble Jar
java -jar *$DUCC_HOME*/lib/ducc-unreserve.jar

Java main
org.apache.uima.ducc.cli.DuccReservationCancel

If no options are given, help text is presented.

Options:

--id [jobid]

The ID is the reservation ID returned by the job submission.

--help

Prints the usage text to the console.

4.5. ducc_monitor

The source for this section is ducc_ducbook/documents/part-user/cli/monitor.xml

Description:

It may be desired to monitor a job's progress after it has been submitted. The monitor CLI connects to the DUCC message flow and provides job status as it progresses including state changes, error counts, and number of work items processed.

Usage:

Script wrapper
$DUCC_HOME/bin/ducc_monitor

Executable Jar
java-jar *$DUCC_HOME*/lib/ducc-monitor.jar

Java main
org.apache.uima.ducc.cli.DuccJobMonitor

If no options are given, help text is presented.

Options:

--cancel_job_on_interrupt

If the monitor is canceled with Ctrl-C, the job is also canceled. Otherwise the monitor is simply disconnected and the job continues.

If *--wait_for_completin* is not specified this option is ignored.

--debug

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

--help

Prints the usage text to the console.

--id [jobid]

The ID is the jobid returned by the job submission.

--timestamp

If specified, messages are timestamped.

4.6. ducc_service_submit

The source for this section is ducc_ducbook/documents/part-user/cli/service_submit.xml

Description:

The *ducc_service_submit* CLI is used to submit a job as a *service* to DUCC. The CLI is similar to *ducc_submit* with the following key differences:

- There is no Collection Reader.
- There is no Job monitor for services because services don't generally end of their own accord.
- Service jobs must supply a fully-formed DD XML.

On submission of a service, the DUCC CLI examines the service DD descriptor for the queue name, and the supplied *jvm_args* for a broker URL. It forms a service ID of the following form which may be referenced in the *--service_dependency* clauses of jobs and services which are dependent on this service:

```
UIMA-AS:[endpoint]:[broker-url]
```

Usage:

Script wrapper
$DUCC_HOME/bin/ducc_service_submit

Executble Jar
java -jar *$DUCC_HOME*/lib/ducc-service-submit.jar

Java main
org.apache.uima.ducc.cli.DuccServiceSubmit

If no options are given, help text is presented.

Options:

--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. Example:

```
--description "This is my very sophisticated job"
```

--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. 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 the user's home directory. 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 is descripbed in Chapter 5, *Job Logs* [47].

> **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_classpath [ClASSPATH]

This specifies the Java CLASSPATH to use in each Job Process (JP) and must be specified. Example:

```
--process_classpath a.jar:b.jar
```

.

--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 CLASSPATH and data path as described in the *notes [26]*. For example:

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

--process_deployments_max [integer]

This specifies the maximum nunber of Job Processes to deploy at any given time.

```
--process_deployments_max 66
```

.

--process_environment [environment]

This specifies environment parameters for the Job Processes. If present, they are added to the Job Process environment as the process is spawned. It must be a quoted, blank-delimeted lsit of name-value pairs. For example:

```
"--process_environment TERM=xterm DISPLAY=:1.0"
```

Note: On Secure Linux systems, the environemnt variable LD_LIBRARY_PATH may not be passed to the user's program. If it is necessary to pass LD_LIBRARY_PATH to the JP or JD processes, it must be specified as DUCC_LD_LIBRARY_PATH. Ducc (*securely*) passes this as LD_LIBRARY_PATH, *after* the JP or JD has assumed the user's identity. For example:

```
"--process_environment TERM=xterm DISPLAY=:1.0 DUCC_LD_LIBRARY_PATH=/my/own/lib.so"
```

--process_failures_limit [integer]

This specifies the maximum number of individual Job Process (JP) failures that are to be tolerated before killing the job. The default is 15. 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 independent Job Process initialization failures (i.e. System.exit(), kill-15...) before the number of Job Processes is capped at the number in state Running currently. The default is 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_jvm_args [list]

This specifies additinal arguments to be passed to the Job Process JVM. Example:

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

--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. if this amount is exceeded by a Job Process the Agent terminates the process with a ShareSizeExceeded message. Example:

```
--process_memory_size 33
```

--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. Example:

```
--schedling_class normal
```

--service_dependency[list]

This specifies a comma-delimeted list of services the job processes are dependent upon. Each endpoint must be of the form UIMA-AS:*endpoint*:*broker_url* where *endpoint* is the UIMA-AS service endpoint and *broker_url* is the ActiveMQ broker URL.

In the example are two dependencies, one with endpoint RandomSleepAE and broker tcp:bluej682:61616, and the other with endpoint OtherEp and broker URL tcp:bluej123:123. Example:

```
--service_dependency UIMA-AS:RandomSleepAE:tcp:bluej682:61616, \
            UIMA-AS:OtherEp:tcp:bluej123:123
```

--specifiecaiton [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).

For example,

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

where the *job.props* contains:

```
working_directory=/Users/challngr/projects/ducc/ducc_test/test/bin
process_failures_limit=20
driver_environment= DUCC_LD_LIBRARY_PATH=/a/other/bogus/path
process_environment=AE_INIT_TIME=10000 DUCC_LD_LIBRARY_PATH=/a/bogus/path
log_directory=/Users/challngr/ducc/logs/
process_initialization_failures_cap=99
process_descriptor_AE=org.apache.uima.ducc.test.randomsleep.FixedSleepAE
process_classpath=/home/bob/projects/ducky-service.jar
description=../simple/jobs/1.job[AE]
process_jvm_args=-Xmx100M -DdefaultBrokerURL=tcp://localhost:61616
scheduling_class=fixed
process_memory_size=15
```

--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. Example:

```
--working_directory /Users/challngr/projects/ducc/ducc_test/bin
```

Notes:

When searching for UIMA XML resource files such as descriptors, DUCC searches both the classpath and the data path according to the following rules:

1. If the resource ends in .xml it is assumed the resource is a file and the path is either an absolute path or a path relative to the specified working directory. If the file is not found the search exits and the job is terminated.
2. If the resource does not end in .xml, DUCC creates a path by replacing the "." separators with "/" and appending ".xml". It then searches the CLASSPATH for the resource as a file.

   If the resource is found in either place the search is successful. Otherwise the search fails and the job is terminated.

   > **Note:** Note that in the current implementation, resources are NOT searched for inside jars in the classpath. Files must be supplied.

4.7. ducc_service_cancel

The source for this section is ducc_ducbook/documents/part-user/cli/cancel.xml

Description:

The *ducc_service_cancel* CLI is used to cancel a *submitted* service. Generally services won't end unless canceled. If this is used against a *registered* service instance, the service manager will usually restart the service. Use *ducc_services stop* to stop a *registered* service.

Usage:

Script wrapper
ducc_service_cancel

Executable Jar
java-jar *$DUCC_HOME*/lib/ducc-service-cancel.jar

Java main
org.apache.uima3.ducc.cli.DuccServiceCancel

If no options are given, help text is presented.

Options:

--id [serviceid]

The ID is the jobid returned by the job submission, also available from the webserver.

--help

Prints the usage text to the console.

4.8. ducc_services

The source for this section is ducc_ducbook/documents/part-user/cli/service_api.xml

Description:

The *ducc_services* CLI is used to manage service registration. It has a number of functions as listed below. Additionally the *ducc_services* CLI wraps *ducc_service_submit* and *ducc_service_cancel* for convenience.

The functions include:

Register
: This *registers* a service with the Service Manager. A registered service is retained by DUCC until it is unregistered.

Unregister
: This *unregisters* a service with the Service Manager. When a service is unregistered DUCC optionally stops the service instance, if any, and discards all knowledge of it.

Start
: The *start* function instructs DUCC to alllocate resources for a service and to start it in those resources. The service remains running until explictly 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 registerd and otherwise.

Modify
: The *modify* function allows some aspectes of a registered service to be updated without re-registereing the service. It optionally alters the running service instances to conform with the updates.

Submit
: Use the **ducc_service_submit** command to submit a service. This is available only through the command-line wrapper.

Cancel
: Use the **ducc_service_cancel** command to cancel a submitted service. This is available only through the command-line wrapper.

Usage:

Script wrapper
: *$DUCC_HOME*/bin/ducc_services

Executble Jar
: java -jar *$DUCC_HOME*/lib/ducc-services.jar

Java main
: org.apache.uima.ducc.cli.DuccServiceApi

The **ducc_services** CLI requires one of the verbs listed above as the first argument. The subsequent arguments are determned by the verb.

4.8.1. ducc_service --register

The source for this section is ducc_ducbook/documents/part-user/cli/services_register.xml

Description:

This *registers* a service with the Service Manager. A registered service is retained by DUCC until it is unregistered.

Usage:

See the DUCC Service CLI Overview [34] for general usage considerations for **ducc_services**.

Service Register Options:

--register [properties file] [override options]

The properties file is optional. It is a standard Java properties files containing all the registration options for the service. The override options are then applied to define the service (taking precedence). It is possible to register a service using just a properties file, just override options, or both.

The properties in the properties file are identical to the command-line parameters, but with the leading "--" removed. For example:

```
process_environment = DUCC_LD_LIBRARY_PATH=/my/own/lib.so
description = Test Service 0
process_jvm_args = -Xmx100M -DdefaultBrokerURL=tcp://bluej291:61617
process_classpath = ../../lib/ducc-test.jar
process_memory_size = 15
working_directory = /home/bob/service-descriptors
process_DD = Service_FixedSleep_0.xml
process_deployments_max = 1
scheduling_class = fixed
```

--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. Example:

```
--description "My totaly rad service"
```

.

--help

Prints the usage text to the console.

--instances [number-of-instances]

This defines the default number of service instances to start. If not specified, the default is 1. Example:

```
--instances 12
```

.

--jvm [path-to-java]

States the JVM to use. If not specified, the same JVM used by the Agents is used. 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 the user's home directory. 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 is descripbed in Chapter 5, *Job Logs* [47].

--process_classpath [ClASSPATH]

This specifies the Java CLASSPATH to use in each Job Process (JP) and must be specified. Example:

```
 --process_classpath a.jar:b.jar:more.jar
```

--process_DD [DD descriptor]

This specifies a UIMA Deployment Descriptor (DD) for the service.

--process_environment [environment]

This specifies environment parameters for the Job Processes. If present, they are added to the Job Process environment as the process is spawned. It must be a quoted, blank-delimeted lsit of name-value pairs. For example:

```
"--process_environment TERM=xterm DISPLAY=:1.0"
```

--process_failures_limit [integer]

This specifies the maximum number of individual Job Process (JP) failures that are to be tolerated before killing the job. The default is 15. 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 independent Job Process initialization failures (i.e. System.exit(), kill-15...) before the number of Job Processes is capped at the number in state Running currently. The default is 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_jvm_args [list]

This specifies additinal arguments to be passed to the Job Process JVM. Example:

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

--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. if this amount is exceeded by a Job Process the Agent terminates the process with a ShareSizeExceeded message. Example:

```
--process_memory_size 33
```

--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. Example:

```
--schedling_class normal
```

Note: Note that in general one should select a non-preemptable class such as *fixed>* or *reserve* for services. Otherwise DUCC may grow or shrink the number of processes used by the service. It IS legal and supported to use a *fair-share* class however.

--service_custom_classpath [CLASSPATH]

This specifies the classpath to be used when starting a CUSTOM ping thread. It is primarily intended for non-UIMA-AS services but it may be implemented for UIMA-AS services as well if the default DUCC ping function is not sufficient. Example:

```
--service_custom_classpath A.jar:B.Jar:C.Jar
```

--service_custom_endpoint [CUSTOM:string]

This provides the name of the endpoint to used for non-UIMA-AS services. In the current release of DUCC this type of service must be started independently of DUCC but DUCC is able to monitor it if **--service_custom_ping** is provided. The endpoint must start with the characters "CUSTOM:" followed by any unique string (with no embedded blanks) that DUCC can use to identify the service. Example:

```
--service_custom_endpoint CUSTOM:jrc.service.endpoint
```

--service_custom_jvm_args [list]

This supplies extra arguments to the JVM for the CUSTOM ping object. Example:

```
--service_custom_jvm_args -Xmx 400M -Xms100M
```

--service_custom_ping [java class]

This supplies the java class name for a CUSTOM ping object. The class must the interface **org.apache.uima.ducc.IServiceMeta** as described in the API section. DUCC wraps the customer ping object in a management object with a "main" and calls the implemented interfaces periodically to insure the custom service is functioning, and to gather performance statistics. Example:

```
--service_custom_ping bob.net.BobsCustomPing
```

--service_dependency[list]

This specifies a comma-delimeted list of services the job processes are dependent upon. Each endpoint must be of the form UIMA-AS:*endpoint*:*broker_url* where *endpoint* is the UIMA-AS service endpoint and *broker_url* is the ActiveMQ broker URL.

In the example are two dependencies, one with endpoint RandomSleepAE and broker tcp:bluej682:61616, and the other with endpoint OtherEp and broker URL tcp:bluej123:123. Example:

```
--service_dependency UIMA-AS:RandomSleepAE:tcp:bluej682:61616, \
                UIMA-AS:OtherEp:tcp:bluej123:123
```

--service_linger [time in seconds]

This specifies the time, in seconds, that a service should be kept alive after its last reference has exited, in anticipation of new work entering the system and using it. This is only applicable to services that are not automatically started at boot time. Example:

```
--service_linger 300
```

--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. Example:

```
--working_directory /Users/challngr/projects/ducc/ducc_test/bin
```

4.8.2. ducc_services --start

The source for this section is ducc_ducbook/documents/part-user/cli/service_start.xml

Description:

The **start** function instructs DUCC to alllocate resources for a service and to start it in those resources. The service remains running until explictly 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.

Usate:

See the DUCC Service CLI Overview [34] for general usage considerations for **ducc_services**.

Service Start Options:

--start [service id]

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 endpoing 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, the registered number of instances is started. If the number 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 ar started for service 33 for a totoal of ten. The registry is updated only if the *--update* option is also specified.

Example:

```
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
```

4.8.3. ducc_services --stop

The source for this section is ducc_ducbook/documents/part-user/cli/service_stop.xml

Description:

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 [34] to stop submitted services.

Usage:

See the DUCC Service CLI Overview [34] for general usage considerations for **ducc_services**.

Service Stop Options:

--stop [service id]

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.

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 a number is specified, then only the specified number of instances are stopped. Thus if ten instances are running and *ducc_services --stop 33 --instances 5* is issued, five (randomly selected) service instances ar stopped for service 33, leaving five running. The registry is updated only if the *--update* option is specified. The registered number of instances is never reduced to 0.

Example:

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

--update

If specified, the registry is updated to the total number of instances remaining, but is never reduced below 1.

Example:

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

4.8.4. ducc_services --modify

The source for this section is ducc_ducbook/documents/part-user/cli/service_modify.xml

Description:

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

Usage:

See the DUCC Service CLI Overview [34] for general usage considerations for **ducc_services**.

Service Modify Options::

--modify [service id]

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.

Example:

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

4.8.5. ducc_services --query

The source for this section is ducc_ducbook/documents/part-user/cli/service_query.xml

Description:

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.

Usage:

See the DUCC Service CLI Overview [34] for general usage considerations for **ducc_services**.

Service Query Options:

--query [service id]

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.

Example: below is a query against a system with three services.

The service with endpoint **UIMA-AS:FixedSleepAE_6:tcp://bluej291:61617** is a service submitted outside of DUCC so it is marked as *Internal* and has no implementing processes tha are known to DUCC. It is used by job 0 and is active, available, and being actively pinged. The ActiveMq queue statistics are shown.

The service with endpoint **UIMA-AS:FixedSleepAE_5:tcp://bluej291:61617** is a registered service, whose registered numeric id is **2**. It is registered for two instnaces 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 is currently used (referenced) by DUCC jobs 1 and 5.

The service with endpoint **UIMA-AS:FixedSleepAE_1:tcp://bluej291:61617** is a submitted service. It was submitted twice, and so has two implementors, DUCC service jobs 0 and 1. It is referenced by job 7. It will continue to run until somebody cancels it, even if it is not used.

```
Service: UIMA-AS:FixedSleepAE_6:tcp://bluej291:61617
   Service Class    : Implicit
   Implementors     : (N/A)
   References       : 0
   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
        78     240    170       2   36414      0      0     636     806     636
```

```
Service: UIMA-AS:FixedSleepAE_5:tcp://bluej291:61617
   Service Class    : Registered as ID 2 instances[2] linger[5]
   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

Service: UIMA-AS:FixedSleepAE_1:tcp://bluej291:61617
   Service Class    : Submitted
   Implementors     : 1 0
   References       : 7
   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       0      0      1 1504371      0      0      35      35      35
```

4.8.6. ducc_services --submit and --cancel

The source for this section is ducc_ducbook/documents/part-user/cli/service_sub_can.xml

Description:

As a convenience, both ducc_service_submit and ducc_service_cancel can be invokded from ducc_services. **Ducc_services** is just a thin wrapper around those two commands.

Usage:

See the DUCC Service CLI Overview [34] for general usage considerations for **ducc_services**.

Service Submit Options:

--submit [parameters]

The parameters are the same parameters as for ducc_service_submit.

Example:

```
                    ducc_services --submit --specification 123.service
```

--cancel [parameters]

The parameters are the same parameters as for ducc_service_cancel.

Example:

```
ducc_services --cancel --id 4
```

Chapter 5. Job Logs

The source for this chapter is ducc_ducbook/documents/part-user/userlogs.xml

The DUCC logs are managed by *log4j* and are configured using `ducc_runtime/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/1.2/.

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. The logs and stdout written by the UIMA collection reader. The collection reader uses the UIMA logger which is by default directed to stdout.
2. The diagnostic logs written the the DUCC JD wrapper around the job's collection reader. This log is written using *log4j*.

A number of other usefiles 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 written to `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 UIMA-AS 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 Java serialized object containing the performance breakdown for the job. This is used by the Web Server to display the breakdown. This file is written to `job-performance-summary.ser`.

The JP logs are written by default to `HOME/ducc/logs`, where HOME is the submitting user's home directory. In this directory, a subdirectory whose name is the numerical id of the job is created by DUCC, where all logs for the job are written.

The collection reader's log is written to the file `HOME/ducc/logs/JOBID/jd.out.log` via *log4j*. It is written in multiple generations, and its size is governed by the same *log4j* configuration file used for the DUCC Daemon processes. The size of each generation and the number of generations is configured in the *jdout* appender stanza.

Each JP log and the diagnostic JD log is 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 runs.

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

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

```
100-JD-bluej290-1-29383.log
100-uima-ae-descriptor-29383.xml
100-uima-as-dd-29383.xml
100-UIMA-bluej290-2-32766.log
100-UIMA-bluej291-63-13594.log
jd.out.log
job-performance-summary.ser
job-specification.properties
```

In this example,

The file `100-JD-bluej290-1-29383.log` is the diagnostic JD log, where the JD executed on node bluej290-1 in process 29383.

The file `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.

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

The file `100-UIMA-bluej290-2-32766.log` is a JP log for job 100, that ran on node bluej290-2, in process 32766.

The file `100-UIMA-bluej291-63-13594.log` is a JP log for job 100, that ran on node bluej291-63, in process 13594

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

The file `job-performance-summary.ser` is the serialized performance breakdown that is displayed in the Web Server

The file `job-specification.propeties` is the properties file describing the job.

Chapter 6. Application Programming Interface (API)

The source for this chapter is ducc_ducbook/documents/chapter-api.xml

There is a partial DUCC API. Completion of the API is planned for the next major update and will not be documented until the design and first implementation is complete.

Chapter 7. Webserver

The source for this chapter is ducc_ducbook/documents/chapter-webserver.xml

The DUCC Web Server default address is accessed from the URL `http://wshost:42133`. Each local installation configures the host for "wshost" and may override the default port of *42133*

The Webserver is designed to be mostly self-documenting. The design is intentionally simple and contains a link to this document. Column headers and reason/state codes have display a short description if you hover your mouse over it.

The columns can all be sorted by clicking on the column headers.

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

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

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

DuccBook
This is a link to the HTML version of the documnet 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 submenu 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 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.

End
This is the time the Job completes.

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. States include:

Received - The jobhas ben vetted, persisted, and assigned a unique ID.
WaitingForDriver - The job is waiting for the Job Driver to initialize.
WaitingForServices - The job is waiting to verify that any declared services are available.
WaitingForResources - The job is waiting to be scheduled.
Initializing - The job is in its initializaiton phase.
Running - At least one process is now initialized and running.
Completing - The last process has finished and the job is cleaning up.
Completed - The job is complete.

Reason
This is information relating to completion state.

EndOfJob - The job 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 administator canceled the job.
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.
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. Check the JP logs for the reason.
ProcessFailure - Too many processes failed while running. 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 allocaiton 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.

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 showing the specific failures.

Run Fails
This is the total number of process failures experienced by the job. This field is hyperlinked to a page showing the specific failures.

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 successfuly 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 a page showing the specific failures.

Dispatch
This is the total number CASs that are currently dispatched. This is usally min(Processes * Threads, incomplete_work_items - errors)

Retry
This is the number of CASs that were retried for any reason (such as timeout).

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 descriptin string from the *--description* string from submit.

7.3. Job Details Page

This page shows details of all the processes that run in support of a job.

The Jobs page contains the following columns:

Id
This is the DUCC process id (not the Operating System's processid). Process 0 is alwyas the Job Driver. It is hyperlinked to *jd.out.log*.

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 check directly.

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

PID
This is the Operating Systems' PID for 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
This shows why a process is terminated, from the system's point of view.

AutonomousStop - The process terminated unexpectedly of its own accord ("crashed") for no detectable reason..
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.
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 initialization step failed.
Forced - The node is preempted by RM for other work because of fair share.
InitializationTimeout - The initialization phase exceeded the configured timeout.
Killed - The agent terminated the process for some reason.
Stopped - The job is winding down, there's no more work for this node, so it stops.
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. Check the JP and JD logs for possible causes..

State:Agent
If there's an error detected only by the agent, this shows the Agent's reason for a process's death.

Reason:Agent
If there's an error detected only by the agent, this shows the Agent's reason for a process's death.

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

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.

%GC
Process percentage of time spent in garbage collections, relative to total of initialization + run times.

CPU
Cumulative CPU time for the process.

%RSS
Resident Storage Size, as a percentage of process memory requirement in job specification.

Time:Avg
Average seconds spent per work item in the process.

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

Time:max
This is the minimum time spent per work item in theprocess.

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 for any reason, excluding preemptions.

Preempt
This is the number of work items that had to be retried because of preemption.

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

7.4. Reservation Details Page

This page shows details of all reservations.

The Reservations page contains the following columns:

Id
This is the DUCC process id of the reservation as provided when the reservation is made.

Start
This is the time the reservation was mde.

End
This is the time the reservation was canceled.

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

Class
This is the resource class used to schedule the reservation.

Status
This is the status of the reservation. Values include:

Received - Reservation has been vetted, persisted, and assigned unique Id.

WaitingForResources - The reservation is waitng for the Resource Manager to find and schedule resources.
Assigned - The reservation is active.
Completed - The reservation has been canceled.

Reason

If a reservation is not active, the reason. Reasons include:

ResourcesUnavailable - The Resource Manager was unable to find free or freeable resources to match the resource request.
CanceledBySystem - The job was canceled because DUCC was shutdown.
CanceledByUser - The owner or administrator released the reservation.

Allocation

The number of resources (shares for FIXED policy reservartions, processes for RESERVE policy reservations) that are allocated.

Size

The memory size in GB of the each allocated unit.

List

The node names of the machines where the resource is allocated.

Description

This is the descriptin string from the *--description* string from submit.

Chapter 8. Examples: Building and Testing a Simple Application

The source for this chapter is ducc_ducbook/documents/chapter-webserver.xml

This chapter intentionally left blank.

To hold you over until this chapter is filled in, the complete source for the sample jobs is installed into `ducc_runtime/test/src`.

Part III. DUCC Administration Guide

Chapter 9. Installation, Configuration, and Verification

The source for this chapter is ducc_ducbook/documents/chapter-install.xml

This chapter describes how to install, configure, and verify DUCC.

In this document we refer to the machines in a DUCC cluster as the "worker" machines (or nodes) and the "administrative" machines (or nodes). Applications are distributed to the "worker" nodes. The DUCC processes which manage resources, process deployment, web serving, etc, are run on the "administrative" nodes.

In secure environments it may be desirable to run both the "worker" and "administrative" processes behind a firewall, inaccessible to the public at large. In this case it is possible to configure the DUCC web-server to run on a gateway machine. We thus may refer to the node with the DUCC web-server as the "web-server" node.

9.1. General Considerations

DUCC should be installed on systems *dedicated* to running DUCC and applications managed by DUCC. DUCC is designed to manage applications that are highly memory-intensive. The DUCC Resource Manager assumes that every processor in the cluster is dedicated to a single instance of the DUCC Agent and its spawned children. Prohibitively high levels of page / swap activity may result from sharing processors with DUCC, preventing applications from making progress and in worst cases, locking out the processors.

9.2. Hardware Requirements

The following are *minimal* hardware requirements for running DUCC.

- One Intel-based or IBM Power-7 system. DUCC clusters may be heterogeneous, composed of both Intel and Power hardware.
- Eight GB (8GB) RAM on each system. DUCC support heterogeneous memory sizes across all configured processors.

9.3. Software Requirements

The following are minimal software requirements for DUCC. This software must be installed on all DUCC nodes.

- A modern Linux system. DUCC has been developed and tested on SUSE and Debian distributions.
- IBM or Sun JRE 1.6 or greater. DUCC has only been tested on 1.7 JREs.
- Python 2.x where "x" is at least 4. The oldest version of Python supported by DUCC is 2.4. DUCC has not been tested under any version of Python 3. All modern Linux distributions supply an acceptable version by default. It may be necessary for the System Administrator to install Python from the Linux distribution media as it is not installed in some default configurations.

- User and group "ducc" must be established on all machines. For security reasons, the group "ducc" should not be shared with any other users.

  Currently user "ducc" is hard-coded into the security code of DUCC and cannot be changed. It is possible to run DUCC under any userid. However all jobs will run under the identity of the user starting DUCC. This is acceptable for testing and system verification but is a potential security problem for general use.

- All machines in the DUCC cluster must be connected via a shared file system and a shared user space. DUCC assumes all user home directories as well as the "ducc" home directory are cross mounted on all machines.

- Password-less ssh must be installed on the JD and worker machines for user id "ducc".

- At least one user id other than "ducc" that is available to all nodes, to submit jobs from.

  Note: User "root" cannot be used to submit jobs to DUCC. User "ducc" should not be used to submit jobs.

9.4. Quick Installation Checklist

Note: Throughout this document the location where DUCC is installed is referred to as *ducc_runtime*. By default, the installation procedures install DUCC in the home directory of user *ducc* as *~ducc/ducc_runtime* where *~ducc* refers to ducc's home directory.

This is an overview of the installation and verification procedures. Details for this checklist follow in the next section.

1. Configure user ducc and group ducc on all systems.
2. Expand the distribution tarfile.
3. Run the installation script *ducc_install*.
4. Install the utility *ducc_ling* on local disk space and set permissions.
5. Update *ducc_runtime/resources/ducc.properties*:
   - Specify location of installed *ducc_ling*.
   - Specify the correct ActiveMQ broker address.
   - Specify location of the installed JRE.
   - Configure the HTTP hostname and optionally, the HTTP port for the Orchestrator.
   - Configure the HTTP hostname and optionally, the HTTP port for the Service Manager.
   - Optionally specify the node for the DUCC webserver.
6. Create node configuration ducc.nodes in ducc_runtime/resources.
7. Optionally update the file ducc_runtime/resources/reserved.nodes.
8. Optionally create or update the file ducc_runtime/resources/ducc.administrators.

9. Run the verify_ducc utility, repeating and correcting problems, until no errors are reported.
10. Start the ActiveMQ broker and ensure it is running.
11. Start DUCC.
12. From a web browser, go to the URL http://ducchost:42133 and ensure the machines and DUCC daemons are present and running, where *ducchost* is the nodename where the browser is started.
13. Run the verification procedures.

9.5. Detailed Installation Procedures

This section provides detail instructions for installing DUCC.

9.5.1. Basic System Initialization

Create a user "ducc" and a group "ducc". Currently the user and group must both be "ducc". This ID is hard-coded into the *ducc_ling* utility for security reasons.

Ensure Python 2.x is installed as the "default" Python. DUCC has only been tested on Python version 2.4 and 2.6. It may not work on Python 3.0.

Ensure that the IBM or SUN JRE 1.6 is installed on every node. The full JDK is only needed on nodes where applications are being developed. The location of this JRE must be coded into *ducc.properties* as described below and is used to run the DUCC processes. It is possible for applications to use different JREs via the job specifications.

9.5.2. Install DUCC Distribution

Log in as user *ducc* and expand the DUCC distribution file:

```
tar -zxf [ducc-distribution-file].tgz
```

This creates a directory *ducc=distribution-0.6.4-beta* with the installation materials.

Now execute the installation scripting:

```
cd ducc-distribution-0.6.4-beta
./ducc_install
```

You will be prompted for the location of the ducc_runtime and ActiveMQ installations. First-time users should take the defaults and simply hit *enter* at each prompt.

This will create and populate two directories:

```
~ducc/activemq - the ActiveMQ distribution
~ducc/ducc_run-time - the DUCC run-time
```

Installation also ensures all necessary programs are made executable and it installs the ActiveMQ configuration that is tested and customized for DUCC.

> **Note:** It is possible to use an existing ActiveMQ broker instead of the one supplied with DUCC as long as it is fully compatible with ActiveMQ 5.5. If this is desired, enter NONE at the prompt for the ActiveMQ location. Be aware that careful tuning of the ActiveMQ broker may be necessary to support both the DUCC load and the existing load however.

9.5.3. Perform Post-Installation Tasks

This section describes how to configure DUCC and secure the *ducc_ling* utility.

9.5.3.1. *ducc_ling*

Ducc_ling is a setuid-root program that DUCC uses to spawn jobs under the identity of the submitting user. To do this, *ducc_ling* must briefly acquire *root* privileges in order to switch to the user's identity. *ducc_ling* itself takes care not to open any security holes while doing this but it must be correctly installed to prevent malicious or errant processes from compromising system security.

There are three points to make about *ducc_ling*, described in detail below:

1. *Ducc_ling* must be carefully secured to avoid accidental breach of security by setting ownership and file permissions correctly.
2. It is possible to run *ducc_ling* without root privileges, albeit with some restrictions of DUCC function.
3. *Ducc_ling* may need to be rebuilt for your hardware.

Securing *ducc_ling*

To secure *ducc_ling*, it must be installed on local disk space (not on a shared file system), on all of the DUCC nodes. The necessary procedure is to create a directory dedicated to containing *ducc_ling* and set the privileges on that directory so only user *ducc* is able to access its contents.

Next, copy *ducc_ling* into the local, now protected, directory, and set its privileges and ownership so that when it executes, it executes as user *root*. When invoked, *ducc_ling* immediately assumes the identity of the job owner, sets the working directory for the process, establishes log directories for the job, and execs into the specified job process.

The following steps illustrate how to do this. Root authority is needed to perform these steps. If local procedures prohibit the use of setuid-root programs, or root authority cannot be obtained, it is still possible to run DUCC; however,

1. All jobs will then run as user *ducc* as it will be impossible for them to assume the submitter's identity.
2. File-system permissions must be set for all DUCC users so that user *ducc* is able to read their applications and data during execution.

For the sake of these procedures, assume that *ducc_ling* is to be installed on local disk in the directory:

```
/local/ducc/bin
```

Ducc_ling is supplied in the installation directory as

```
ducc_runtime/admin/ducc_ling
```

Remember that this procedure must be performed *as root* on *every* node in the DUCC cluster.

1. Create the directory to contain *ducc_ling*:

   ```
   mkdir /local
   mkdir /local/bin
   mkdir /local/ducc/bin
   ```

2. Ensure that */local/ducc/bin* has correct permissions, allowing only the *ducc* user to read, write, or execute its contents.

   ```
   chown ducc.ducc /local/ducc/bin
   chmod 700 /local/ducc/bin
   ```

3. Copy *ducc_ling* into place:

   ```
   cp ducc_runtime/admin/ducc_ling /local/ducc/bin
   ```

4. Set ownership of *ducc_ling*. It is necessary to ensure that user ownership is *root* and that group ownership is *ducc*.

   ```
   chown root.ducc /local/ducc/bin/ducc_ling
   ```

5. Set permisisons so that user *root* can read, write, and execute *ducc_ling*, group *ducc* can read and execute, and that when *ducc_ling* is executed, it is run as the user who owns it (the "setuid" bit).

   ```
   chmod 4750 /usr/bin/ducc/ducc_ling
   ```

When done correctly, only user ducc will have the ability to access ducc_ling. ducc_ling has internal checks to prevent it from operating when invoked by root and to prevent it from executing jobs as user root. Assuming ducc_ling is installed in /local/ducc/bin, the ducc_ling permissions should be as follows (the date and file-sizes will not match this example):

```
ducc@f7n1:~/ducc-0.1-beta> ls -l /local/ducc/bin
-rwsr-x--- 1 root ducc 22311 2011-10-08 11:42 ducc_ling
```

NOTE the **-rwsr-x---** permissions on ducc_ling. If this is not what you see then retry the procedure.

Running *ducc_ling* Without Root Authority

It is possible to run DUCC without giving *ducc_ling* root authority if there are security concerns or simply if you wish to experiment with DUCC on a machine where you cannot get (or do not want) root privileges. If you do this, all jobs will execute under the identity of the user that starts DUCC. For example, if you install DUCC and start it as user "bob", then all jobs run as user "bob". Most of DUCC is developed and tested in this mode and it is expected to work correctly.

To run *ducc_ling* in this mode , simply use the default configuration as distributed in ducc.properties, and the DUCC agents will use the non-privileged version instead. *Ducc_ling* will execute from the directory

```
ducc_runtime/admin
```

This is very convenient for running small test systems or for simply evaluating DUCC before performing a more extensive installation.

The default configuration line for ducc_ling to run in this mode is as follows:

```
ducc.agent.launcher.ducc_spawn_path=${DUCC_HOME}/admin/ducc_ling
```

Notes:

- If you run in this mode, you do NOT need to install ducc_ling in local disk space; the *ducc_ling* that is packaged in *ducc_runtime/admin* will work.
- If *ducc_ling* is compiled for an architecture other than the one you installed in, you will need to rebuild it for your architecture as described below.

Running On Architectures Other Than That In The Prebuilt Distribution.

DUCC is almost a pure-Java application. However a small bit of C code called ducc_ling is required to allow DUCC to assume different user's identity. Your tarball will come with ducc_ling compiled for some specific architecture. To build ducc_ling for a different architecture (e.g. Intel, Power, or other), all that is needed is normal gcc.

To rebuild ducc_ling:

- CD to the directory with the ducc_ling source:

  ```
  cd ducc_runtime/duccling/src
  ```

- Build ducc_ling:

  ```
  make clean all
  ```

When done you have an architecture-specific ducc_ling binary that must be installed as described above.

9.5.4. Update ducc.properties

The file *ducc.properties* is the main configuration file for DUCC. Some properties must not be changed or DUCC will not function; these properties control internal DUCC operations. Other properties are tuning parameters that should not be adjusted until experience with the local installation is gained the the tuning requirements are known. Some properties define the local environment and must be set when DUCC is first installed.

The properties that must be updated as part of installation are:

```
ducc.broker.hostname
ducc.broker.port
ducc.jvm
ducc.ws.node
ducc.ws.address
ducc.sm.http.port
ducc.sm.http.node
ducc.orchestrator.http.port
ducc.orchestrator.node
ducc.agent.launcher.ducc_spawn.path
```

The full set of properties is described in ducc.properties [77]

Edit ducc_runtime/resources/ducc.properties and adjust the required properties as follows:

ducc.broker.hostname
: Set this to the host where your ActiveMQ broker is running. This MUST be set to the host-name, not "localhost", even if your broker port is configured to "localhost" or "0.0.0.0". There is no default for this parameter.

ducc.broker.port
: Set this to the port configured for ActiveMQ. The default is 61616.

ducc.jvm
: Set this to the full path to the "java" command on your systems. If this is not set DUCC will attempt to use the "java" command in its path and will fail if this is not the correct version of java, or if it is not in the default path.

  Note that Java must be installed on all nodes in the same location. For example:

  ducc.jvm = /share/bin/jdk1.6/bin/java

ducc.ws.node
: Set this to the node name where you want your web-server to run. If not set, the web-server starts on the same node as the rest of the DUCC management processes.

ducc.ws.address
: In multi-homed systems (more than one network card), the DUCC web-server will not know which address it should listen on for requests. Set this address to the desired web-server address. If the system is not multi-homed this property need not be set.

ducc.sm.http.port
: This is the HTTP port for SM requests. The default is 19989. If this is acceptable, it may be left as is; otherwise, select a port and configure it here.

ducc.sm.http.node
: This MUST be configured to the node where the SM is running. The default is a placeholder, "localhost", which will not generally work.

ducc.orchestrator.http.port
: This is the HTTP port for most commands (ducc_submit, ducc_reserve, etc.) The default is 19988. If this is acceptable, it may be left as is; otherwise, select a port and configure it here.

ducc.orchestrator.node
: This MUST be configured to the node where the Orchestrator is running. The default is a placeholder, "localhost", which will not generally work.

ducc.agent.launcher.ducc_spawn.path
: Set this to the full path where *ducc_ling* is installed.

9.5.5. Create the DUCC Node list

Update the file "ducc.nodes" in the directory "ducc_runtime/resources/". For initial installation this should be a simple flat file with the name of each host that participates in the DUCC cluster on one line. The section on ducc.nodes [108] provides full details on node configuration. Note that line comments are allowed and are denoted with #. For example:

```
# Frame 6 nodes
f6n6           # management node
f6n7
f6n8
f6n9
f6n10
# Frame 7 nodes
f7n1
# Frame 10 nodes
f10n1
f10n2
f10n3
f10n8
f10n9
```

Note: It is important that the node running the management processes is NOT in the nodelist. If the management node is in the nodelist an agent will be started on that node and Job Processes (JPs) will be started on it. Because JPs use a very large amount of memory this can prevent the management processes from functioning.

9.5.6. Define the Job Driver nodepool

One node should be defined for running the Job Driver (JD) processes. This may be any node in the cluster. The node must be reserved to prevent Job Processes (JP) from running on it. It is permissible for the JD reserved node to be the management node, as long as sufficient memory (at least 16GB) is available. To constrain the Job Driver node to a specific set of nodes, it is necessary to define a nodepool containing those nodes, and to update the JobDriver class to use that node pool. Details on nodepool and class configuration are in ducc.classes [105].

If it doesn't matter which node is reserved for the Job Driver this step may be skipped.

Configure the Job Driver node thus:

1. Create the file *ducc_runtime/resources/jobdriver.nodepool*
2. Add the name of the management node to the file. This should be the only line in the file.
3. Configure the *JobDriver* class in *ducc.properties* to be in the jobdriver nodepool.

For example:

```
bash-3.2$ cat jobdriver.nodepool
f6n6    # management and job driver node
```

9.5.7. Define the system administrators

Userids listed in file *ducc_runtime/resources/ducc.administrators* are granted expanded privileges, for example the ability to cancel any job on the system via the DUCC web-server. The format of the file is simply one userid per line, with commented lines denoted by a leading #. For example:

```
# administrators
degenaro
challngr
cwiklik
eae
```

9.6. Run The Verification Script

The script ~*ducc/ducc_runtime/admin/verify_ducc* checks your ActiveMQ configuration, ducc.nodes, and ducc_ling setup to ensure the steps above were completed correctly.

Simply execute the script, fixing problems and rerunning until no errors are reported. If ANY errors are reported they must be fixed and *verify_ducc* rerun before continuing.

```
cd ducc_runtime/admin
./verify_ducc
```

9.7. Start DUCC

You should add the directory `ducc_runtime/admin` to your path to simplify DUCC administration. As well you should add `ducc_runtime/bin` to your path in order to submit and cancel jobs and reservations.

1. Start the ActiveMQ broker. If you're using the broker supplied with DUCC use the following procedure, otherwise use your local procedures.

   ```
   cd ~ducc/activemq/apache-activemq-5.5.0/bin
   ```

```
./activemq start
```

2. Ensure the broker is running. If you use the ActiveMQ distribution supplied with DUCC and are using the default port, then use the following command, otherwise use your local procedures

   ```
   netstat -an | grep 61616 | grep LISTEN
   ```

   You should see something similar to the following if ActiveMQ is started correctly. Be sure ActiveMQ is started before continuing (because ActiveMQ manages all message flows and acts as the DUCC name server.)

   ```
   tcp46      0      0  *.61616      *.*        LISTEN
   ```

3. Start DUCC. The command below starts DUCC using the default node list, *ducc.nodes.* See the section describing *start_ducc* for other options.

   ```
   cd ~ducc/ducc_runtime/admin
   ./start_ducc
   ```

4. Make sure DUCC is running on all the expected nodes by running the *check_ducc* script. You would expect to see a process for each of

   - rm: the Resource manager
   - sm - the Services manager
   - pm - the Process manager
   - ws - the web-server
   - or - the job flow manager

   and you would expect to see one agent on each node specified in ducc.nodes.

   For example:

   ```
   ducc@f10n1:~/projects/ducc/ducc_build/runtime/admin> ./check_ducc
   Checking f10n1 ... Found rm @ f10n1 PID 95288 owned by ducc
   Found pm @ f10n1 PID 95337 owned by ducc
   Found sm @ f10n1 PID 95409 owned by ducc
   Found or @ f10n1 PID 95478 owned by ducc
   Found agent @ f10n1 PID 95621 owned by ducc
   Checking f10n2 ... Found agent @ f10n2 PID 92113 owned by ducc
   Checking f10n3 ... Found agent @ f10n3 PID 58602 owned by ducc
   Checking f10n4 ... Found agent @ f10n4 PID 31689 owned by ducc
   Checking f10n5 ... Found agent @ f10n5 PID 122128 owned by ducc
   Checking f10n6 ... Found agent @ f10n6 PID 8301 owned by ducc
   Checking f10n7 ... Found agent @ f10n7 PID 106659 owned by ducc
   ```

```
Checking f10n8 ... Found agent @ f10n8 PID 43946 owned by ducc
Checking f10n9 ... Found agent @ f10n9 PID 115101 owned by ducc
Checking f10n10 ... Found agent @ f10n10 PID 93730 owned by ducc
Checking f9n2 ... Found ws @ f9n2 PID 88351 owned by ducc
```

9.8. Start DUCC Browser

Open a browser to the URL http://wshost:42133, where "wshost" is the host where the DUCC webserver is started in the previous step. Feel free to explore.

Click the "Status" and then "Machines" link at the upper left to see the machines that are configured above. If they do not show up after a minute or two there is something wrong with the installation.

Click the "Status" and then "Reservations" link. This should show a reservation for user "System" and class "JobDriver". The status should show "Assigned" or "Waiting For Resources". If it shows "Waiting For Resources" it may take two to three minutes to advance to "Assigned". If it never becomes "Assigned" there is something wrong with the installation.

Once the machines and JobDriver reservation show up correctly DUCC is ready to run work.

9.9. Run a Job

> **Note:** Jobs cannot be scheduled until all DUCC components have initialized and stabilized, which can take a minute or two. Check the web console, under Status -> Reservations and wait until the reservation for JobDriver is in state "Assigned" before attempting to run jobs.

A set of very simple jobs is provided in the distribution for testing and demonstration. The jobs are installed into `ducc_run-time/test` as part of the installation above. The jobs run UIMA analytics but instead of computation, they simply sleep, in order to verify and demonstrate DUCC without the need for high-powered hardware and complex software installation.

To run a job:

1. Set your path to include *ducc_runtim/bin*. This directory has all the commands for the Command Line Interface (CLI).

2. As some user other than *ducc*, go to the directory `ducc_runtime/test/jobs` and run *ducc_submit*:

   ```
   cd ~ducc/ducc_runtime/test/jobs
   ducc_submit --specification 1.job
   ```

   A job id number is printed to the console.

It will take a few moments for resources to be scheduled and the job to start up. You can follow the progress of the job in the web browser using the *Status* -> *Jobs* link.

In your home directory expect to find the following:

- The directory `ducc/logs` is created.

- Inside `ducc/logs` is a directory with the same id as was given when you submitted should appear. As the job progresses, a number of logs and other files will be created in this directory.

There are five sample jobs provided, each of which runs a different number of work items, and one which submits a *reservation*. One need not wait for one job to complete before submitting another; try submitting several of the jobs and watch progress on the web-server and visualization.

You may cancel any job while it is running by executing *ducc_cancel:*

```
ducc_cancel --id [id]
```

where the ID you supply is the one returned by *ducc_submit*. The ID is also shown in the web-server.

To submit a reservation:

```
ducc_reserve --specification reserve.job
```

This will take a few moments and if all is well, will return an ID. The reservation will have been scheduled when the ID is returned. It is possible to view the reservation in the web-server under Status -> Reservations.

To cancel the reservation:

```
ducc_unreserve --id [id]
```

again, using the ID returned from *ducc_reserve.*

The commands issued here and the format of the inputs are described in detail in the Command Line Interface chapter.

9.10. Shutdown DUCC

To stop DUCC, execute

```
~ducc/ducc_runtime/admin/ducc_stop -a
```

This broadcasts a message to all DUCC processes instructing them to terminate. Any job processes still alive are also killed.

Shutdown attempts to be "graceful". If there is still a job running a signal is sent indicating shutdown is occurring and DUCC waits a few moments for processes to exit. If the processes do not exit DUCC issues *kill -9* to forcibly stop them, and then exits.

Occasionally system problems prevent a DUCC process from stopping. It is good practice, after stopping DUCC, to ensure the processes actually exited by running

```
check_ducc
```

check_ducc searches all the nodes in the node list and the local node for DUCC processes and prints a status line for everything it finds.

If, after a minute or two, *check_ducc* shows some DUCC process still running, you can have *check_ducc* issue *kill -9* against them:

```
check_ducc -k
```

Chapter 10. Administration

The source for this chapter is ducc_ducbook/documents/chapter-admin.xml

This chapter describes how to start, stop, and generally administer DUCC.

There are several files used to configure DUCC:

ducc.properties
: *Ducc.properties* contains primary configuration for all the DUCC processes including URL and port specifications, tuning parameters, and protocols.

ducc.classes
: *Ducc.classes* configures the scheduling classes used by the Resource Manager.

Node lists
: *Node lists* contain the names of the machines that comprise a DUCC cluster. The default nodelist is called *ducc.nodes*

Node pools
: *Nodepool files* are used by the Resource Manager to logically group nodes by function. Jobs, reservations, and services may be restricted to specific pools of nodes.

These are several administrative commands. To use these one should add *ducc_runtime/admin* to their PATH. It is not necessary to add DUCC_HOME to the environment as the commands infer its location from where they are invoked.

start_ducc
: Use *start_ducc* to start DUCC administrative processes and agents. The DUCC processes may be started individually or at once.

stop_ducc
: Use *stop_ducc* to stop DUCC administrative processes and agents. The DUCC processes may be stopped individually or at once.

check_ducc
: Use *check_ducc* to query the state of DUCC processes in the cluster. It may be used to force-kill DUCC processes and user processes that cannot be stopped by *stop_ducc* for any reason.

verify_ducc
: Use *verify_ducc* validate the integrity of your DUCC installation. It performs extensive checks to insure components are installed as expected.

10.1. ducc.properties

The source for this chapter is ducc_ducbook/documents/admin/ducc-properties.xml

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. Changing these parameters may create imbalances in the system and result in performance problems or even prevent DUCC from operating at all. In the description below these properties are classified as "Tuning".

Some of the properties are standard configuration properties: 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".

10.1.1. General DUCC Properties

ducc.jms.provider

Default Value
: activemq

Type
: Private

Purpose
: Declare the type of middleware providing the JMS service used by DUCC.

ducc.broker.protocol

Default Value
: tcp

Type
: Private

Purpose
: Declare the wire protocol used to communicate with ActiveMQ.

ducc.broker.hostname

Default Value
: localhost

Type
: Local

Purpose
: This declares the node name 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.

ducc.broker.port

Default Value
: 61616

Type
Local

Purpose
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.

ducc.broker.decoration

Default Value
wireFormat.maxInactivityDuration=0

Type
Local

Purpose
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.

ducc.broker.name

Default Value
localhost

Type
Local

Purpose
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". When using another broker, this name must match the "brokerName" parameter in the local ActiveMQ configuration.

ducc.broker.jmx.port

Default Value
1099

Type
Local

Purpose
This is the port used to make JMX connections to the broker. When using the ActiveMQ broker supplied with DUCC this should normally not be changed. If using another ActiveMQ broker this must match the configured JMX port.

ducc.cluster.name

Default Value
Welcome To DUCC!

Type
Local

Purpose
This is a string used in the Web Server banner to identify the local cluster. It may be set to anything desired.

ducc.runmode

Default Value
unconfigured.

Type
Local

Purpose
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 multi-user environment without the need for root privileges.

> **Note:** WARNING! Enabling this feature in a production DUCC system is a serious security breach. It should only be set by DUCC developers running with an un-privileged *ducc_ling*.

ducc.locale.language

Default Value
en

Type
Private

Purpose
Establish the language for national language support of messages. Currently only "en" is supported.

ducc.locale.country

Default Value
us

Type
Private

Purpose
Establish the country for National Language Support of messages. Currently only "us" is supported.

ducc.jvm

Default Value
java

Type
: Local

Purpose
: Specifies the full path to the JVM to be used by the DUCC processes. If not specified, "java" must be in the default path for user "ducc".

ducc.jmx.port

Default Value
: 2099

Type
: Private

Purpose
: 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.

ducc.agent.jvm.args

Default Value
: Xmx100M

Type
: Tuning

Purpose
: This specifies the list of arguments passed to the JVM when spawing the Agent.

ducc.orchestrator.jvm.args

Default Value
: Xmx1G

Type
: Tuning

Purpose
: This specifies the list of arguments passed to the JVM when spawing the Orchestrator.

ducc.rm.jvm.args

Default Value
: Xmx1G

Type
: Tuning

Purpose
This specifies the list of arguments passed to the JVM when spawing the Resource Manager.

ducc.agent.jvm.args

Default Value
Xmx1G

Type
Tuning

Purpose
This specifies the list of arguments passed to the JVM when spawing the Process Manager.

ducc.sm.jvm.args

Default Value
Xmx1G

Type
Tuning

Purpose
This specifies the list of arguments passed to the JVM when spawing the Service Manager.

ducc.ws.jvm.args

Default Value
Xmx8G

Type
Tuning

Purpose
This specifies the list of arguments passed to the JVM when spawing the Webserver.

ducc.admin.endpoint

Default Value
ducc.admin.channel

Type
Private

Purpose
This is the JMS endpoint name used for DUCC administration messages.

ducc.admin.endpoint.type

Default Value
topic

Type
Private

Purpose
: This is the JMS message type used for DUCC administration requests. If changed DUCC admin may not work.

ducc.submit.threads.limit

Default Value
: (unconfigured)

Type
: Local

Purpose
: This enforces a maximum number of threads per job, amortized over all the processes. No job will have more threads than this dispatched. This limit is disabled by default.

ducc.driver.jvm.args

Default Value
: (unconfigured)

Type
: Local

Purpose
: If enabled, the arguments here are automatically added to the JVM arguments specified for the Job Driver process.

ducc.process.jvm.args

Default Value
: (unconfigured)

Type
: Private

Purpose
: If enabled, the arguments here are added by DUCC to the JVM arguments in the user's job processes.

ducc.cli.httpclient.sotimeout

Default Value
: 0

Type
: Tuning

Purpose
: This is the timeout used by the CLI to communicate with DUCC, in millisseconds. 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.

ducc.signature.required

Default Value
: on

Type
: Tuning

Purpose
: When set, the CLI signs each request so the Orchestrator can be sure the requestor is actually who he claims to be.

10.1.2. Web Server Properties

ducc.ws.configuration.class

Default Value
: org.apache.uima.ducc.ws.config.WebServerConfiguration

Type
: Private

Purpose
: The name of the pluggable java class used to implement the Web Server.

ducc.ws.node

Default Value
: (unconfigured)

Type
: Local

Purpose
: This is the name of the node the web server is started on. If not specified, the web server is started on the node where *start_ducc* is run.

ducc.ws.ipaddress

Default Value
: (unconfigured)

Type
: Local

Purpose
: 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.

ducc.ws.port

Default Value
: 42133

Type
: Local

Purpose
: This is the port on which the DUCC Web Server listens for requests.

ducc.ws.port.ssl

Default Value
42155

Type
Local

Purpose
This is the port that the Web Server uses for SSL requests (such as authentication).

ducc.ws.port.ssl.pw

Default Value
quackquack

Type
Local

Purpose
This is the SSL password used for SSL requests.

ducc.ws.session.minutes

Default Value
60

Type
Local

Purpose
Once authenticated, this property determines the lifetime of the authenticated session to the Web Server.

ducc.ws.max.history.entries

Default Value
200

Type
Local

Purpose
The Web Server maintains a history of jobs over time. To avoid overloading the system with data about old and obsolete jobs it prunes the history. This property determines the size of the history that is kept.

10.1.3. Job Driver Properties

ducc.jd.configuration.class

Default Value
org.apache.uima.ducc.jd.config.JobDriverConfiguration

Type
Private

Purpose
The name of the pluggable java class used to implement the Job Driver.

ducc.jd.state.update.endpoint

Default Value
ducc.jd.state

Type
Private

Purpose
This is the JMS endpoint name by the Job Driver to send state to the Orchestrator.

ducc.jd.state.update.endpoint.type

Default Value
topic

Type
Private

Purpose
This is the JMS message type used to send state to the Orchestrator.

ducc.jd.state.publish.rate

Default Value
15000

Type
Tuning

Purpose
The frequency in milliseconds that JD publishes its state to the Orchestrator. A higher rate may slightly increase system response but will increase network load. A lower rate will somewhat decrease system response and lower network load.

ducc.jd.queue.prefix

Default Value
ducc.jd.queue.

Type
Private

Purpose
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.

ducc.jd.host.class

Default Value
JobDriver

Type
Tuning

Purpose
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*.

ducc.jd.host.description

Default Value
Job Driver

Type
Tuning

Purpose
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.

ducc.jd.memory.size

Default Value
8GB

Type
Tuning

Purpose
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.

ducc.jd.number.of.machines

Default Value
1

Type
Tuning

Purpose
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.

ducc.jd.host.user

Default Value
System

Type
Tuning

Purpose
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.

10.1.4. Service Manager Properties

ducc.sm.configuration.class

Default Value
org.apache.uima.ducc.sm.config.JobDriverConfiguration

Type
Private

Purpose
This is the name of the pluggable java class used to implement the Service Manager.

ducc.sm.state.update.endpoint

Default Value
ducc.sm.state

Type
Private

Purpose
This is the JMS endpoint name used for state messages sent by the Service Manager.

ducc.sm.state.update.endpoint.type

Default Value
topic

Type
Private

Purpose
This is the JMS message type used for state messages sent by the Service Manager.

ducc.sm.meta.ping.rate

Default Value
60000

Type
Tuning

Purpose
This is the time, in milliseconds, between pings by the Servic Manager to each known, running service.

ducc.sm.meta.ping.stability

Default Value
10

Type
Tuning

Purpose
This is the number consecutive pings that may be missed before a service is considered unavailable.

ducc.sm.meta.ping.timeout

Default Value
5000

Type
Tuning

Purpose
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.

ducc.sm.http.port

Default Value
19989

Type
Local

Purpose
This is the HTTP port used by the SM to field requests from the CLI / API.

ducc.sm.http.node

Default Value
localhost

Type
Local

Purpose
This is the node where the service manager runs. It MUST be configured as part of DUCC setup.

ducc.sm.default.linger

Default Value
300

Type
Tuning

Purpose
This is the length of time, in seconds, that the SM allows a service to remain alive after all referenceing jobs have exited. If no new job enters the system by the time this time has expired, the SM stops the service.

10.1.5. Orchestrator Properties

ducc.orchestrator. configuration.class

Default Value
org.apache.uima.ducc.orchestrator.config.OrchestratorConfiguration

Type
Private

Purpose
This is the name of the pluggable java class used to implement the DUCC Orchestrator.

ducc.orchestrator.checkpoint

Default Value
on

Type
Private

Purpose
This controls Orchestrator state checkpointing. If set off, no state is saved across restarts of the Orchestrator except for the current job numbering. This should generally be left *on*.

ducc.orchestrator.start.type

Default Value
warm

Type
Tuning

Purpose
This indicates the level of recovery to be taken on restarting a system. In general, if DUCC is fully shutdown, only cold and warm starts make sense because the Job Processes and Job Drivers are terminated during the shutdown. However if a management process died or was terminated by the administrators, most work can be recovered without interruption, allowing for a hot start. There are three level 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 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 reservations and services are preserved.
- **hot**. All reservations are restored. The system attempts to reattach to all jobs that are still running. The system attempts to reattach to any services that are still running. Any services that need to be restarted are restarted.

ducc.orchestrator.state.endpoint

Default Value
ducc.orchestrator.state

Type
Private

Purpose
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.

ducc.orchestrator.state. update.endpoint.type

Default Value
topic

Type
Private

Purpose
This is the JMS endpoint type used for the "full" state messages sent by the Orchestrator.

ducc.orchestrator.state.publish.rate

Default Value
15000

Type
Private

Purpose
This is the frequency in milliseconds that the Orchestrator publishes its non-abbreviated state.

ducc.orchestrator.abbreviated. state.endpoint

Default Value
ducc.orchestrator.abbreviated.state

Type
Private

Purpose
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.

ducc.orchestrator.abbreviated. state.update.endpoint.type

Default Value
topic

Type
Private

Purpose
This is the JMS endpoint type used for the "abbreviated" state messages sent by the Orchestrator.

ducc.orchestrator.abbreviated. state.publish.rate

Default Value
15000

Type
Private

Purpose
This is the frequency in milliseconds that the Orchestrator publishes its abbreviated state.

ducc.orchestrator.maintenance.rate

Default Value
60000

Type
Tuning

Purpose
This is the frequency in milliseconds that the Orchestrator checks and updates history and state.

ducc.orchestrator.job.factory.classpath.order

Default Value
user-before-ducc

Type
Tuning

Purpose
When the DUCC Agent spawns a process it must set the process's Java CLASSPATH. This CLASSPATH must contain a minimum set of entries, which are supplied by the Agent. However, users may want their own CLASSPATH to take precedence; for example, they may have a different version of some *.jar* file. In this case the user's CLASSPATH should be set before DUCC's. To control this, set this tuning parameter to one of two values:

- user-before-ducc
- ducc-before-user

10.1.6. Resource Manager Properties

ducc.rm.configuration.class

Default Value
org.apache.uima.ducc.rm.config.ResourceManagerConfiguration

Type
Private

Purpose
This is the name of the pluggable java class used to implement the DUCC Resource Manager.

ducc.rm.state.update.endpoint

Default Value
ducc.rm.state

Type
Private

Purpose
This is the name of the JMS endpoint through which the Resource Manager broadcasts its abbreviated state.

ducc.rm.state.update.endpoint.type

Default Value
topic

Type
Private

Purpose
This is the JMS endpoint type used for state messages sent by the Resource Manager..

ducc.rm.state.publish.rate

Default Value
60000

Type
Tuning

Purpose
This is the rate, in milliseconds, at which the Resource Manager publishes its state to the Orchestrator.

ducc.rm.share.quantum

Default Value
15

Type
Tuning

Purpose
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.*

The job's declared *process_memory_size* is used to determine the overall memory requirements in terms of share quanta according to the formula: physical_requirement = ciel(process_memory_size / share_quantum) * share_quantum.

For example suppose a process declares it's memory requirement to be 20GB. Then physical_requirement = ciel(20 / 15) * 15 = 2 * 15 = 30 GB. The processes for this job are scheduled only on machines with at least 30 GB of reported RAM, and the Resource Manager insures that no other processes are scheduled on the machine that might encroach on this 30 GB.

The *share quantum* is also used to determine each user's *fair share* of the resources. The scheduler's goal is to ensure that all user's are allocated the same number of quantum shares. Conceptually, the total memory in the system is divided by the *share quantum* and then allocated in equal portions to all users in the system.

Thus, jobs that require less memory will generally have more processes scheduled than jobs that require more memory, but the total memory scheduled is approximately the same for all jobs.

ducc.rm.scheduler

Default Value
org.apache.uima.ducc.rm.scheduler.NodepoolScheduler

Type
Private

Purpose
The component that implements the scheduling algorithm is pluggable. This specifies the name of that class.

ducc.rm.class.definitions

Default Value
ducc.classes

Type
Tuning

Purpose
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].

ducc.rm.default.tasks

Default Value
10

Type
Tuning

Purpose
In order to calculate the number of processes to allocate to a job, the scheduler must know how many tasks or *work items* the job will execute. If the job does not declare that number, *default.tasks* is used.

ducc.rm.default.memory

Default Value
15

Type
Tuning

Purpose
If a job does not declare the amount of memory each process requires, the scheduler uses *default.memory* for scheduling. The unit is GB.

Note that the Agents enforce the declared memory, so if a process understates its requirements it will generally be killed.

ducc.rm.default.threads

Default Value
4

Type
Tuning

Purpose
Each job process will be dispatched with some number of threads such that DUCC will dispatch *work items* to these threads. The scheduler uses this number to calculate the number of processes that must be allocated.

The maximum number of processes a job requites is determined by the formula: num_processes = ciel(num_work_items / num_threads).

Thus, a job that declares 100 work items and 4 threads is assigned a maximum of ciel(100/4) = 25 processes.

ducc.rm.node.stability

Default Value
5

Type
Tuning

Purpose
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.

ducc.rm.init.stability

Default Value
3

Type
Tuning

Purpose
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.

ducc.rm.eviction.policy

Default Value
SHRINK_BY_INVESTMENT

Type
Tuning

Purpose
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.

ducc.rm.initialization.cap

Default Value
2

Type
Tuning

Purpose
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 [105]*.

ducc.rm.expand.by.doubling

Default Value
true

Type
Tuning

Purpose
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 [105]*.

ducc.rm.prediction

Default Value
: true

Type
: Tuning

Purpose
: 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 [105]*.

ducc.rm.prediction.fudge

Default Value
: 10000

Type
: Tuning

Purpose
: 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 [105]*.

10.1.7. Agent Properties

ducc.agent.configuration.class

Default Value
: org.apache.uima.ducc.nodeagent.config.AgentConfiguration

Type
: Private

Purpose
This is the name of the pluggable java class used to implement the DUCC Agents.

ducc.agent.request.endpoint

Default Value
ducc.agent

Type
Private

Purpose
This is the JMS endpoint through which agents receive state from the Process Manager.

ducc.agent.request.endpoint.type

Default Value
topic

Type
Private

Purpose
This is the JMS endpoint type used for state messages sent by the Process Manager.

ducc.agent.managed.process. state.update.endpoint

Default Value
ducc.managed.process. state.update

Type
Private

Purpose
This is the JMS endpoint used to communicate from the managed process to the Agent (Job Process).

ducc.agent.managed.process. state.update.endpoint.type

Default Value
socket

Type
Private

Purpose
This is the JMS endpoint type used to communicate from the managed process (Job Process) to the Agent.

ducc.agent.managed.process. state.update.endpoint.params

Default Value
transferExchange=true&sync=false

Type
Private

Purpose
These are configuration parameters for the Agent-to-JP communication socket. These should only be modified by DUCC developers.

ducc.agent.node. metrics.endpoint

Default Value
ducc.node.metrics

Type
Private

Purpose
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.

ducc.agent.node. metrics.endpoint.type

Default Value
topic

Type
Private

Purpose
This is the JMS endpoint type used to send node metrics updates from the agents.

ducc.agent.node. metrics.publish.rate

Default Value
60000

Type
Tuning

Purpose
This is the rate at which node metrics updates are published in milliseconds.

This value MUST be coordinated with the Orchestrator publish rate and the Resource Manager publish ratio. The rate must be at least *ducc.rm.state.publish.ratio* * *ducc.orchestrator.state.publish.rate*. In the default configuration the *ducc.rm.state.publish.ratio* is 4 and the *ducc.orchestrator.state.publish.rate* is 15 seconds, so the *ducc.agent.node.metrics.publish.rate* must be st least 60 seconds or 60000 milliseconds.

Failure to set this correctly may result in incorrectly reported missed heartbeats.

ducc.agent.node. inventory.endpoint

Default Value
ducc.node.inventory

Type
Private

Purpose
: 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.

ducc.agent.node.inventory. endpoint.type

Default Value
: topic

Type
: Private

Purpose
: This is the JMS endpoint type used to send node inventory updates from the agents.

ducc.agent.node. inventory.publish.rate

Default Value
: 10000

Type
: Tuning

Purpose
: This is the rate at which node inventory updates are published in milliseconds.

  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.

ducc.agent.node. inventory.publish.rate.skip

Default Value
: 30

Type
: Tuning

Purpose
: This is the number of times the agent will bypass publishing its node inventory if the inventory has not changed.

ducc.agent.launcher. thread.pool.size

Default Value
: 10

Type
: Tuning

Purpose
: This is establishes the size of the agent's threadpool used to manage spawned processes.

ducc.agent.launcher. use.ducc.spawn

Default Value
: true

Type
: Private

Purpose
: 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.

ducc.agent.launcher. ducc_spawn_path

Default Value
: ${DUCC_HOME}/admin/ducc_ling

Type
: Tuning

Purpose
: 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*.

ducc.agent.launcher. process.stop.timeout

Default Value
: 60000

Type
: Tuning

Purpose
: 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.

ducc.agent.launcher. process.init.timeout

Default Value
: 7200000

Type
: Tuning

Purpose
: 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 InitializationTimout status is send to the job driver (JD) which decides whether to retry the process or terminate the job.

  Note that it is normal for the types of processes that DUCC is designed for to have very long initialization times.

ducc.agent.launcher. share.size.fudge.factor

Default Value
: 5

Type
Tuning

Purpose
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*.

ducc.agent.rogue.process.user.exclusion.filter

Default Value
root,posstfix,ntp,nobody,daemon,100

Type
Tuning

Purpose
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.

ducc.agent.rogue.process.exclusion.filter

Default Value
sshd:,-bash,-sh,/bin/sh,/bin/bash,grep,ps

Type
Tuning

Purpose
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.

10.1.8. Process Manager Properties

ducc.pm.configuration.class

Default Value
org.apache.uima.ducc.pm.config.ProcessManagerConfiguration

Type
Private

Purpose
This is the name of the pluggable java class used to implement the DUCC Process Manager.

ducc.pm.request.endpoint

Default Value
ducc.pm

Type
Private

Purpose
This is the endpoint through which process manager receive state from the Orchestrator.

ducc.pm.request.endpoint.type

Default Value
queue

Type
Private

Purpose
This is the JMS endpoint type used for state messages sent by the Orchestrator.

ducc.pm.state.update.endpoint

Default Value
ducc.pm.state

Type
Private

Purpose
This is the endpoint through which process manager sends its heartbeat. The main receiver is the Web Server for it's daemon status page.

ducc.pm.state.update.endpoint.type

Default Value
topic

Type
Private

Purpose
This is the JMS endpoint type used for process manager heartbeats. The primary receiver is the Web Server for its daemon status page.

ducc.pm.state.publish.rate

Default Value
25000

Type
Private

Purpose
This is the rate at which the process manager publishes its heartbeat, in milliseconds.

10.1.9. Job Process Properties

ducc.uima-as.configuration.class

Default Value
org.apache.uima.ducc.agent.deploy.uima.UimaAsServiceConfiguration

Type
Private

Purpose
This is the name of the pluggable java class that implements the the UIMA-AS service shell for job processes (JPs).

ducc.uima-as.endpoint

Default Value
ducc.job.managed.service

Type
Private

Purpose
This is the endpoint through which job processes (JPs) receive messages from the Agents.

ducc.uima-as.endpoint.type

Default Value
socket

Type
Private

Purpose
This is the JMS endpoint type used for messages sent to the JPs from the Agents.

ducc.uima-as.endpoint.params

Default Value
transferExchange=true&sync=false

Type
Private

Purpose
This configures the JP-to-Agent communication socket. It should be changed only by DUCC developers.

ducc.uima-as.saxon.jar.path

Default Value
file:${DUCC_HOME}/lib/saxon8/saxon8.jar

Type
Private

Purpose
This configures the path the required Saxon jar.

ducc.uima-as.dd2spring.xsl.path

Default Value
${DUCC_HOME}/admin/dd2spring.xsl

Type
Private

Purpose
This configures the path the required dd2spring xsl definitions.

ducc.uima-as.flow_controller.specifier

Default Value
org.apache.uima.ducc.uima.DuccJobProcessFC

Type
Private

Purpose
This configures the pluggable class that implements the default flow controller used in the DUCC job processes (JPs).

10.2. ducc.classes

The source for this chapter is ducc_ducbook/documents/admin/ducc-classes.xml

The class configuration file is used by the Resource Manager configure the rules used for job scheduling. See the Resource Manager chapter for a detailed description of the DUCC schedueler.

The name of class configuration file is specified in *ducc.properties*. The default name is *ducc.classes [105]* and is specified by the property *ducc.rm.class.definitions* property.

This file configures the classes and the associate scheduling rules of each class. It contains properties to declare the following:

1. The names of each class.
2. The default class to use if none is specified with the job.
3. The names of all the nodepools.
4. For each nodepool, the name of the file containing member nodes.
5. A set of properties for each class, declaring the rules enforced by that class.

The general properties are as follows. The default values are the defaults in the system as initially installed.

scheduling.class.set

Default Value
: background low normal high urgent weekly fixed reserve JobDriver

Purpose
: This lists the names of classes defined in the file.

scheduling.default.name

Default Value
: normal

Purpose
: This is the default class that jobs are assigned to, when not otherwise designated in their submission properties.

Nodepools are declared with a set of properties to name each nodepool and to name a file for each pool that declares membership in the nodepool. For each nodepool a property of the form *scheduling.nodepool.NODEPOOLNAME* is declared, where *NODEPOOLNAME* is one of the declared nodepools.

The property to declare nodepool names is as follows:

scheduling.nodepool

Default Value
: reserve

Purpose
: This is the list of nodepool names. For example:

```
scheduling.nodepool = res res1 res2
```

This is an example of a declaration of three nodepools.

```
scheduling.nodepool       = res res1 res1
scheduling.nodepool.res   = res.nodes
scheduling.nodepool.res1 = res1.nodes
scheduling.nodepool.res2 = res2.nodes
```

There is no way to enforce priority assignment to any given nodepool. It is possible to declare a "preference", such that the resources in a given nodepool are considered first when searching for nodes. To configure a preference, use the **order** decorattion on a nodepool specificaion.

To declare nodepool order, specify

scheduling.nodepool.[poolname].order. The nodepools are sorted numerically according to their order, and pools with lower order are searched before pools with higher order. The global nodepool always order "0" so it is usally searched first. For example, the pool configuration below establishes a search order of

1. global

2. res2
3. res
4. res1

This is an example of a declaration of three nodepools.

```
scheduling.nodepool             = res res1 res1
scheduling.nodepool.res         = res.nodes
scheduling.nodepool.res.order  = 4
scheduling.nodepool.res1        = res1.nodes
scheduling.nodepool.res1.order = 7
scheduling.nodepool.res2        = res2.nodes
scheduling.nodepool.res2.order = 2
```

For each class named in *scheduling.class.set* a set of properties is specified, defining the rules implemented by that class. Each such property is of the form

```
scheduling.class.CLASSNAME.RULE = VALUE
```

where

CLASSNAME
: This is the name of the class.

RULE
: This is the name of the rule. Rules are described below.

VALUE
: This is the value of the rule, as described below.

The rules are:

policy
: This is the scheduling policy, required, and must be one of:

FAIR_SHARE
FIXED_SHARE
RESERVE

share_weight
: This is any integer. This is the weighted-fair-share weight for the class as discussed above. It is only used when policy = FAIR_SHARE.

priority
: This is the evaluation priority for the class as discussed above. This is used for all scheduling policies.

cap
: This is an integer, or an integer with "%" appended to denote a percentage. It is used for all scheduling classes.

This is the class cap as discussed above. It may be an absolute value, in *processes* (which may comprise more than one share quanta), or it may be specified as a percentage by appending

"%" to the end. When specified as a percentage, it caps the shares allocated to this class as that percentage of the total shares *remaining when the class is evaluated.*. It does not consider shares that may have been available and assigned to higher-priority classes.

nodepool
This is the name of the nodepool associated with this class. It must be one of the names declared in the property *scheduling.nodepool*.

prediction
Acceptable values are *true* and *false*. When set to *true* the scheduler uses prediction when allocating shares. It is only used when policy = FAIR_SHARE.

prediction.fudge
Acceptable values are any integer, denoting milliseconds. This is the prediction fudge as discussed above. It is only used when policy = FAIR_SHARE.

expand.by.doubling
Acceptable values are *true* and *false*. When set to *true* the scheduler doubles a job's shares up to it's fair-share when possible, as discussed above. It is only used when policy = FAIR_SHARE.

expand.by.doubling
Acceptable values are *true* and *false*. When set to *true* the scheduler doubles a job's shares up to it's fair-share when possible, as discussed above. When set in *ducc.classes* it overrides the defaults from *ducc.properties*. It is only used when policy = FAIR_SHARE.

initialization.cap
Acceptable values are any integer. This is the maximum number of processes assigned to a job until the first process has successfully completed initialization. To disable the cap, set it to zero *0*. It is only used when policy = FAIR_SHARE.

max_processes
Acceptable values are any integer. This is the maximum number of processes assigned to a FIXED_SHARE request. If more are requested, the request is canceled. It is only used when policy = FIXED_SHARE. If set to 0 or not specified, there is no enforced maximum.

max_machines
Acceptable values are any integer. This is the maximum number of machines assigned to a RESERVE request. If more are requested, the request is canceled. It is only used when policy = RESERVE. If set to 0 or not specified, there is no enforced maximum.

enforce.memory
Acceptable values are *true* and *false*. When set to *true* the scheduler requires that any machine selected for a reservation matches the reservation's declared memory. The declared memory is converted to a number of quantum shares. Only machines whose memory, when converted to share quanta are selected. When set to *false*, any machine in the configured nodepool is selected. It is only used when policy = RESERVE.

10.3. ducc.nodes

The source for this chapter is ducc_ducbook/documents/admin/ducc-nodes.xml

The DUCC node list is used to configure the nodes used to run jobs and assign reservations. A DUCC Agent is started by DUCC on every node in the node 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.

A node list is a simple flat file where each line consists of a single node name or an *import* statement. Nodes may be designated by IP address or by name. The node list may be commented using the comment delimeter "#".

An *import* statement is of the form

```
import filename
```

where "filename" is the name of another node list. The imported nodelist may itself contain *import* statements to allow a nested organization of lists.

When an *import* statement is encountered, the named file is read and its contents appended to the stream of incoming nodes. The list of nodes used by commands is composed of the nodes in the first list and all imported files.

Examples:

```
cat ducc.nodes

# First four nodes
ducc01.local.net            # 64 GB
ducc02.local.net            # 64 GB
ducc03.local.net            # 128 GB
ducc04.local.net            # 128 GB
import big.nodes
```

```
cat big.nodes

# Large memory nodes, all with 256 GB
ducc11.local.net
ducc12.local.net
ducc13.local.net
ducc14.local.net
```

10.4. Nodepool Configuration

The source for this chapter is ducc_ducbook/documents/admin/ducc-nodepool.xml

Nodepool files are constructed identically to *ducc.nodes [108]*. Nodes may legally occur in either

No nodepool whatever. In this case, the node is considered a member of the default*global nodepool.*
Exactly ONE nodepool file. No node may be a member of more than one nodepool.

The nodepool file is read when the Resource Manager initializes. Every node that checks-in with the RM is then associated with one of the configured nodepools, or with the global nodepool.

10.5. start_ducc

The source for this chapter is ducc_ducbook/documents/admin/start-ducc.xml

Description:

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_runtime/resources/ducc.nodes. This is the equivalent of

```
start_ducc -n $DUCC_HOME/resources/ducc.nodes -m
```

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
```

-m, --management

Start the management processes (rm, sm, pm, orchestrator) on the local node. The webserver is started on the local node, or the node configured in *ducc.properties.*

-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.

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 all DUCC processes, using custom nodelists:

```
start_ducc -m -n foo.nodes -n bar.nodes
```

Start just management processes:

```
start_ducc -m
```

Start just agents on a specific set of nodes:

```
start_ducc -n foo.nodes -n bar.nodes
```

Start and agent on a specific node:

```
start_ducc -c agent@a.specific.node
```

Start the webserver on node 'bingle':

```
start_ducc -c ws@bingle
```

10.6. stop_ducc

The source for this chapter is ducc_ducbook/documents/admin/stop-ducc.xml

Description:

Stop_ducc is used to stop DUCC processes. If run with no parameters it takes the following actions the help text is printed to the console.

Usage:

ducc_stop [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.*

```
stop_ducc -a
```

-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
```

-m, --management

Stop only the management processes *rm, pm, or, sm, and ws.* All agents are left running; all job drivers are left running, all job processes are left running.

-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.

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.

agent
Node Agents.

-k, --kill

Use this to forcibly kill a component using *kill -9*. This should only be used if the *-a* option does not work. This normally has the same effect as *check_ducc -k*, with the difference that *check_ducc* indiscriminately kills all the DUCC processes it can find, whereas *stop_ducc-k* can be directed to a specific instance of a component.

10.7. check_ducc

The source for this chapter is ducc_ducbook/documents/admin/check-ducc.xml

Description:

Check_ducc is used to find and report on DUCC processes. It can be used to find processes owned by ducc (management processes, agents, and job processes), or ducc jobs owned by 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 -n ../resources/ducc.nodes
```

This searches for all the processes owned by user *ducc* on all the nodes in ducc.nodes. User processes are not searched for.

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
```

-u --user [userid]

The userid specifies the user whose processes check_ducc searches for. If not specified, the user executing check_ducc is used. If the user is specified as 'all' then all ducc processes belonging to all users are searched for.

```
check_ducc -u billy
```

-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 `ducc_runtime/state/ducc.pids`.

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.

-r --reap

Reap user processes. This uses kill -9 and ducc_ling to forcibly terminate user processes. Only processes specified by '-u' or '--userid' are targeted. If the user "all" is specified, then all user processes are terminated. The intent of this is to easily find and terminate "rogue" user processes that do not terminate.

Use this option with care. It does not distinguish user processes by specific job id. Every process started by DUCC owned by the designated user is killed.

```
check_ducc -u billy -u bobby -r
```

10.8. verify_ducc

The source for this chapter is ducc_ducbook/documents/admin/verify-ducc.xml

Description:

verify_ducc performs a number of internal consistency checks to insure the DUCC installation is complete and has no obvious configuration errors. The following checks are performed:

- Insure *ducc_ling* is installed in the configured location and has correct permission and ownership on all nodes.
- Insure ActiveMQ is installed and configured in a way compatible with the ActiveMQ URL in ducc.properties.
- Insure all nodelists exist and are readable.
- Insure all nodes can be reached via ssh.
- Insure all nodes are running identical versions of DUCC.
- Insure java is installed in the location configured in ducc.properties on all nodes.
- Print the version of java on all nodes.
- Print the version of operating system on all nodes.
- Print the amount of RAM on all nodes.
- Insure all configured nodepools in ducc.classes exist and reference nodes are configured in the nodelists.
- Insure that all nodepools referenced by classes also defined.

Usage:

verify_ducc [options]

If no options are given, the nodes in `ducc_runtime/resources/ducc.nodes` are used and the default ActiveMQ broker location of `~ducc/activemq` is used.

Options:

-b [broker_install_dir]

This specifies the name of the ActiveMQ broker configuration file that you are using.

```
verify_ducc -b /home/challngr/amqbroker/amq/conf/activemq-nojournal5.xml
```

-n [nodelist]

This specifies the nodelist against which the DUCC installation is verified. This nodelist should be the same nodelist that DUCC will be started with.

Notes:

It may take a couple attempts to get *verify_ducc* to run without error. It is important that all problems reported by *verify_ducc* are handled before trying to start DUCC the first time.

It is recommended that *verify_ducc* be run after any update to the DUCC configuration, most importantly, the addition of nodes. *Verify_ducc* checks the *ducc_ling* configuration on the new nodes as well as verifies network and *ssh* connectivity.

10.9. Logs

The source for this chapter is ducc_ducbook/documents/admin/logs.xml

There are two sets of DUCC logs:

1. daemon logs
2. user logs

User logs are described in the User's Guide section of this book.

The DUCC logs are managed by *log4j* and are configured using `ducc_runtime/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/1.2/.

The *daemon* logs are written by the DUCC administrative processes and reside in `ducc_runtime/logs`.

The logs are based on log4j's *RollingFileAppender* and are configured to roll over after they reach a certain size to prevent them from overflowing their disk space. When a log file reaches a maximum size it is renamed and a new generation of log is created. The maximum size and the number of generations is configured in `ducc_runtime/log4j.xml`.

To configure the size and number of generations of files for each log, the following parameters, specific to each log, may be updated in `log4j.xml`:

maxBackupIndex
: This is the maximum number of generations of the log that will be created. For example, to configure a maximum of 20 generations:

```
<param name="maxBackupIndex" value="20" />
```

maxFileSize
: This is the maximum size of any generation of log file. For example, to configure the maximum to 20 MB:

```
<param name="maxFileSize" value="20MB" />
```

The relevant *log4j* appenders include:

rmlog
: This is the log for the Resource Manager.

pmlog
: This is the log for the Process Manager.

orlog
: This is the log for the Orchestrator.

smlog
: This is the log for the Service Manager.

wslog
: This is the log for the Web Server.

agentlog
: This is the log for the Agents.

Chapter 11. Resource Management, Operation, and Configuration

The source for this chapter is ducc_ducbook/documents/chapter-resource-manager.xml

11.1. Overview

The DUCC Resource Manager is responsible for allocating cluster resources among the various requests for work in the system. DUCC recognizes three classes of work:

1. *Managed Jobs*. Managed jobs are Java applications implemented in the UIMA framework. They 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.

2. *Services*. Services are long-running processes which perform some function on behalf of jobs or other services. Most DUCC services are UIMA-AS services and are managed the same as *managed jobs*. From a scheduling point of view, there is no difference between services and managed jobs.

3. *Reservations*. A reservation provides persistent, dedicated use of some portion of the resources to a specific user. A reservation may be for an entire machine, or it may be for some portion of a machine. Machines are subdivided according to the amount of memory installed on the machine.

The work that DUCC is designed to support is extremely memory-intensive. In most cases resources are significantly more constrained by memory than by CPU processing poser. The entire resource pool in a DUCC cluster therefore consists of the total memory of all the processors in the cluster.

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 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 (independently within each class), collect all the jobs assigned to that user and apply *fair-share* to equally divide all the user's shares among their 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 or can be made available through preemption of fair-share jobs, the reservation is satisfied and resources are allocated. If not, the reservation fails. In the case where preemptions are required, the reservation is delayed until all necessary resources have been freed.

11.2. 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 treated as a reservation in that it is permanently allocated (until it is canceled) and it cannot be preempted by any other request.

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.

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.

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.

11.3. 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 specify the importance (or weights) of the job classes for use by the weighted fair-share scheduling policy.

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 help 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 goes on to the next class 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 that 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.

Class caps may also be used to insure that very high importance jobs cannot fully monopolize all of the resources in the system.

11.4. 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. The highest level nodepool is called the "global" nodepool. If a job class does not have an associated nodepool, the global nodepool is implicitly associated with he class.

Nodepools are associated with job classes. 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 insures that the nodepool mechanism does not disrupt fair-share allocation.

If it is desired that jobs assigned to some subpool take priority over jobs that have spilled over from the "superpool", then the class associated with the subpool should be given greater weight, or greater priority, as appropriate. (See the Weight vs Priority discussion.)

There is no explicit priority associated with nodepools. However, it is possible to assign a "preference" to a specific nodepool, if it is desired that those nodes be chosen first when the are available. Use the nodepool configurations "order" directive to do this.

11.5. Job Classes

The primary abstraction to control and configure the scheduler is the *class*. A *class* is simply a set of rules used to parameterize how resources are assigned to jobs. Every job that enters the system is associated with one 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.

Caps

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.