Distributed UIMA Cluster Computing

Written and maintained by the Apache
UIMA™ Development Community

Version 2.0.0

Copyright © 2012 The Apache Software Foundation

Copyright © 2012 International Business Machines Corporation

License and Disclaimer The ASF licenses this documentation to you under the Apache License, Version 2.0 (the "License"); you may not use this documentation except in compliance with the License. You may obtain a copy of the License at

http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, this documentation and its contents are distributed under the License on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

Trademarks All terms mentioned in the text that are known to be trademarks or service marks have been appropriately capitalized. Use of such terms in this book should not be regarded as affecting the validity of the the trademark or service mark.

Publication date: August 2015

Table of Contents

I DUCC Concepts 1

1 DUCC Overview 2
- 1.1 What is DUCC? 2
- 1.2 DUCC Job Model 2
- 1.3 DUCC From UIMA to Full Scale-out 3
- 1.4 Error Management 5
- 1.5 Cluster and Job Management 5
- 1.6 Security Measures 6
  - 1.6.1 ducc_ling 7
- 1.7 Security Issues 7

2 Glossary 8

II Ducc Users Guide 10

3 Command Line Interface 11
- 3.1 The DUCC Job Descriptor 11
- 3.2 Operating System Limit Support 12
- 3.3 Command Line Forms 12
- 3.4 DUCC Commands 13
- 3.5 ducc_submit 13
- 3.6 ducc_cancel 17
- 3.7 ducc_reserve 18
- 3.8 ducc_unreserve 19
- 3.9 ducc_process_submit 19
- 3.10 ducc_process_cancel 21
- 3.11 ducc_services 22
  - 3.11.1 Common Options 23
  - 3.11.2 ducc_services –register [specification file] [options] 23
  - 3.11.3 ducc_services –start options 26
  - 3.11.4 ducc_services –stop options 26
  - 3.11.5 ducc_services –enable options 26
  - 3.11.6 ducc_services –disable options 27
  - 3.11.7 ducc_services –observe_references options 27
  - 3.11.8 ducc_services –ignore_references options 27
  - 3.11.9 ducc_services –modify options 27
  - 3.11.10 ducc_services –query options 28
- 3.12 viaducc and java_viaducc 29

4 The DUCC Public API 30
- 4.1 Overview Of The DUCC API 30
- 4.2 Compiling and Running With the DUCC API 31
- 4.3 Java API 31

5 Service Management **32**
5.1 Overview. . . . 32
5.2 Service Types. . . . 33
5.3 Service Instance IDs 33
5.4 Service References and Endpoints 33
5.5 Service Management Policies 34
5.6 Service Pingers 36
5.6.1 The Pinger API 36
5.6.2 Declaring a Pinger in A Service 37
5.6.3 Implementing a Pinger 37
5.6.4 Building And Testing Your Pinger 38
5.6.5 Globally Registered Pingers 40
5.7 Sample Pinger 40
5.7.1 Using the Sample Pinger 40
5.7.2 Understanding Sample Pinger 41
5.7.3 Calculating New Deployments in the Pinger 42
5.7.4 Summary of Sample Pinger 45

6 Job Logs **46**

7 DUCC Web Server **48**
7.1 Common Links 49
7.2 Jobs Page 50
7.3 Job Details Page 53
7.3.1 Processes 53
7.3.2 Work Items 56
7.3.3 Performance 57
7.3.4 Specification 58
7.4 Reservation Page 58
7.5 Managed Reservation Details Page 60
7.5.1 Processes 60
7.5.2 Specification 61
7.6 Services Page 61
7.7 Service Details Page 62
7.7.1 Processes 63
7.7.2 Specification 64
7.8 System Details Page 64
7.8.1 Administration 64
7.8.2 Classes 65
7.8.3 Daemons 65
7.8.4 Machines 65
7.9 Visualization 67

III Programming Model And Applications **68**

8 Building and Testing Jobs **69**
8.1 Overview 69
8.1.1 Basic Job Process Threading Model 69
8.1.2 Alternate Pipeline Threading Model 69
8.1.3 Overriding UIMA Configuration Parameters 70
8.2 Collection Segmentation and Artifact Extraction 70
8.3 CAS Consumer Changes for DUCC 70
8.4 Job Development for an Existing Pipeline Design 70
8.5 Job Development for a New Pipeline Design 71
8.5.1 Collection Reader (CR) Characteristics 71

8.5.2 DUCC built-in Flow Controller . . . 71
8.5.3 Workitem Feature Structure . . . 71
8.5.4 Deployment Descriptor (DD) Jobs . . . 72
8.5.5 Debugging . . . 72

9 Sample Application: Raw Text Processing **73**
9.1 Application Function and Design . . . 73
9.2 Configuration Parameters . . . 73
9.3 Set up a working directory . . . 74
9.4 Download and Install OpenNLP . . . 74
9.5 Get some Input Text . . . 74
9.6 Run the Job . . . 74
9.7 Job Output . . . 75
9.8 Job Performance Details . . . 75

10 Sample Application: CAS Input Processing **77**
10.1 Application Function and Design . . . 77
10.2 Configuration Parameters . . . 77
10.3 Run the Job . . . 77
10.4 Job Performance Details . . . 78
10.5 Limiting Job Resources . . . 78

IV Ducc Administrators Guide **79**

11 Installation, Configuration, and Verification **80**
11.1 Overview . . . 80
11.2 Software Prerequisites . . . 80
11.3 Building from Source . . . 81
11.4 Documentation . . . 81
11.5 Single System Installation and Verification . . . 82
11.6 Minimal Hardware Requirements for Single System Installation . . . 82
11.7 Single System Installation . . . 82
11.8 Initial System Verification . . . 83
11.9 Add additional nodes to the DUCC cluster . . . 84
11.10Ducc_ling Configuration - Running with credentials of submitting user . . . 85
11.11CGroups Installation and Configuration . . . 86
11.12Full DUCC Verification . . . 87
11.13Enable DUCC webserver login . . . 87

12 Administration **88**
12.1 WebServer Authentication . . . 88
12.1.1 Example Implementation . . . 88
12.1.2 IAuthenticationManager . . . 89
12.1.3 IAuthenticationResult . . . 91
12.1.4 Example ANT script to build jar . . . 91
12.1.5 Example ducc.properties entries . . . 91
12.1.6 Example ducc.administrators . . . 92
12.2 Properties . . . 92
12.3 Properties merging . . . 92
12.4 ducc.properties . . . 92
12.4.1 General DUCC Properties . . . 93
12.4.2 Web Server Properties . . . 98
12.4.3 Job Driver Properties . . . 100
12.4.4 Service Manager Properties . . . 101
12.4.5 Orchestrator Properties . . . 103

12.4.6 Resource Manager Properties . . . 104
12.4.7 Agent Properties . . . 107
12.4.8 Process Manager Properties . . . 110
12.4.9 Job Process Properties . . . 111
12.5 ducc.private.properties . . . 112
12.5.1 Web Server Properties . . . 112
12.6 Resource Manager Configuration: Classes and Nodepools . . . 112
12.6.1 Nodepools . . . 113
12.6.2 Class Definitions . . . 116
12.6.3 Validation . . . 119
12.7 Ducc Node Definitions . . . 119
12.8 Ducc User Definitions . . . 120
12.9 Administrative Commands . . . 121
12.9.1 start_ducc . . . 121
12.9.2 stop_ducc . . . 122
12.9.3 check_ducc . . . 124
12.9.4 rm_reconfigure . . . 125
12.9.5 rm_qload . . . 125
12.9.6 rm_qoccupancy . . . 127
12.9.7 vary_off . . . 128
12.9.8 vary_on . . . 128
12.9.9 ducc_properties_manager . . . 128

13 Resource Management 130
13.1 Overview . . . 130
13.2 Preemption vs Eviction . . . 131
13.3 Scheduling Policies . . . 132
13.4 Allotment . . . 132
13.5 Priority vs Weight . . . 132
13.6 Node Pools . . . 133
13.7 Scheduling Classes . . . 133

14 Service Management 135

15 Simulation and System Testing 136
15.1 Cluster Simulation . . . 136
15.1.1 Overview . . . 136
15.1.2 Node Configuration . . . 137
15.1.3 Setting up Test Mode . . . 138
15.1.4 Starting a Simulated Cluster . . . 138
15.1.5 Stopping a Simulated Cluster . . . 139
15.2 Job Simulation . . . 140
15.2.1 Overview . . . 140
15.2.2 Job meta-descriptors . . . 140
15.2.3 *Prepare* Descriptors . . . 141
15.2.4 Services . . . 142
15.2.5 Generating a Job Set . . . 143
15.2.6 Running the Test Driver . . . 143
15.3 Pre-Packaged Tests . . . 144

16 DUCC Web Server Customization 146
16.1 Server Side . . . 146
16.2 Client Side . . . 146
16.3 Build and Install . . . 147

17 Understanding the DUCC logs 148

17.1 Overview . . . 148
17.2 Resource Manager Log (rm.log) . . . 149
17.2.1 Bootstrap Configuration . . . 150
17.2.2 Node Arrival and Missed Heartbeats . . . 151
17.2.3 Node Occupancy . . . 152
17.2.4 Job Arrival and Status Updates . . . 152
17.2.5 Calculation Of Job Caps . . . 153
17.2.6 The “how much” calculations . . . 154
17.2.7 The “what of” calculations . . . 155
17.2.8 Defragmentation . . . 155
17.2.9 Published Schedule . . . 156
17.3 Service Manager Log (sm.log) . . . 158
17.3.1 Bootstrap configuration . . . 158
17.3.2 Receipt and analysis of Orchestrator State . . . 159
17.3.3 CLI Requests . . . 159
17.3.4 Dispatching / Startup of Service Instances . . . 160
17.3.5 Progression of Service State . . . 161
17.3.6 Starting and Logging Pingers . . . 161
17.3.7 Publishing State . . . 162
17.4 (Orchestrator Log or.log) . . . 162
17.5 Process Manager Log (pm.log) . . . 162
17.6 Agent log Log (hostname.agent.log) . . . 162

List of Figures

1.1 Standard UIMA Pipeline . . . 3
1.2 UIMA Pipeline As Scaled by UIMA-AS . . . 4
1.3 UIMA Pipeline As Automatically Scaled Out By DUCC . . . 4
1.4 UIMA Pipeline With User-Supplied DD as Automatically Scaled Out By DUCC . . . 5

5.1 Sample UIMA-AS Service Pinger . . . 38

7.1 Sample Webserver Page . . . 48
7.2 Preferences Page . . . 49
7.3 Jobs Page . . . 53
7.4 Processes Tab . . . 56
7.5 Work Items Tab . . . 57
7.6 Performance Tab . . . 58
7.7 Specification Tab . . . 58
7.8 Reservations Page . . . 60
7.9 Visualization . . . 67

9.1 OpenNLP Process Measurements . . . 76
9.2 OpenNLP Process Breakdown . . . 76

10.1 CAS Input Processing Performacne . . . 78

12.1 Nodepool Example . . . 114
12.2 Nodepools: Overlapping Pools are Incorrect . . . 114
12.3 Nodepools: Multiple top-level Nodepools . . . 115
12.4 Sample Nodepool Configuration . . . 116
12.5 Sample Class Configuration . . . 118
12.6 Sample Node Configuration . . . 120
12.7 Sample User Registration . . . 120

Part I

DUCC Concepts

Chapter 1

DUCC Overview

1.1 What is DUCC?

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

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

1.2 DUCC Job Model

The Job Model defines the steps necessary to scale-up a UIMA pipeline using DUCC. The goal of DUCC is to scale-up any UIMA pipeline, including pipelines that must be deployed across multiple machines using shared services.

The DUCC Job model consists of standard UIMA components: a Collection Reader (CR), a CAS Multiplier (CM), application logic as implemented one or more Analysis Engines (AE), and a CAS Consumer (CC).

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

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

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

- The name of a resource containing the CR descriptor.
- The name of a resource containing the CM descriptor.
- The name of a resource containing the AE descriptor.
- The name of a resource containing the CC descriptor.

- Other information required to parameterize the above and identify the job such as log directory, working directory, desired scale-out, classpath, etc. These are described in detail in subsequent sections.

On job submission, DUCC creates a single process executing the CR and one or more processes containing the analysis pipeline.

DUCC provides other facilities in support of scale-out:

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

1.3 DUCC From UIMA to Full Scale-out

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

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

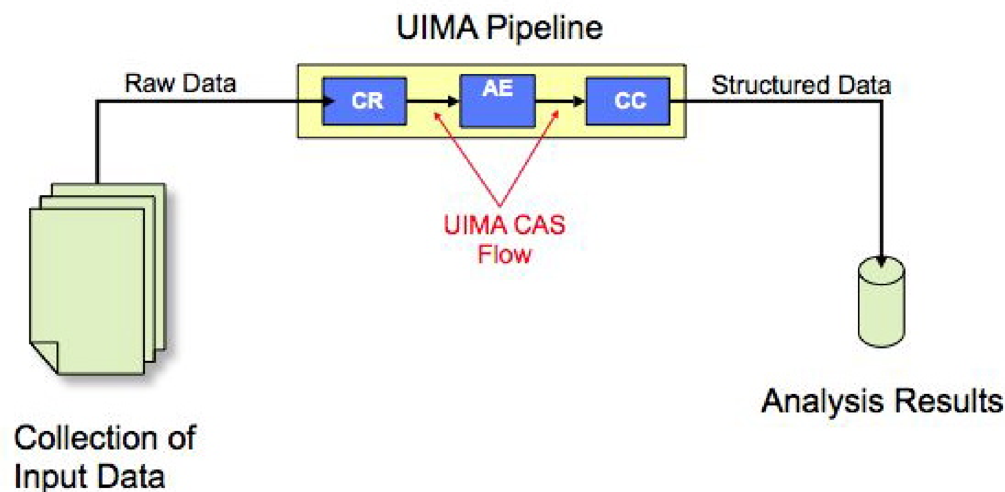

Figure 1.1: Standard UIMA Pipeline

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

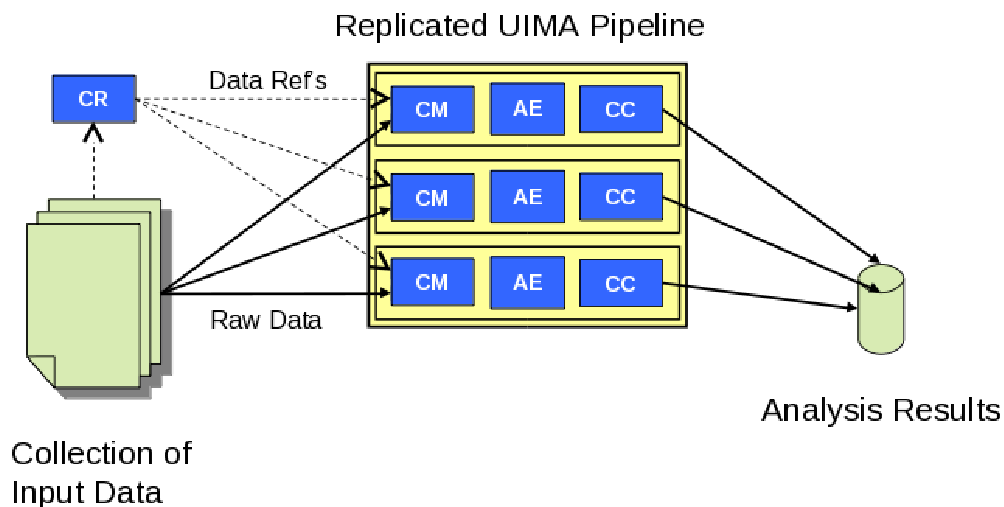

Figure 1.2: UIMA Pipeline As Scaled by UIMA-AS

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

On job submission, the CR is wrapped with a DUCC main class and launched as a Job Driver (or JD). The DUCC main class establishes communication with other DUCC components and instantiates the CR. If the CR initializes successfully, and indicates that there are greater than 0 work items to process, the specified CM, AE and CC components are assembled into an aggregate, wrapped with a DUCC main class, and launched as a Job Process (or JP).

The JP will replicate the aggregate as many times as specified, each aggregate instance running in a single thread. When the aggregate initializes, and whenever an aggregate thread needs work, the JP wrapper will fetch the next work item from the JD, as shown in Figure 1.3 below.

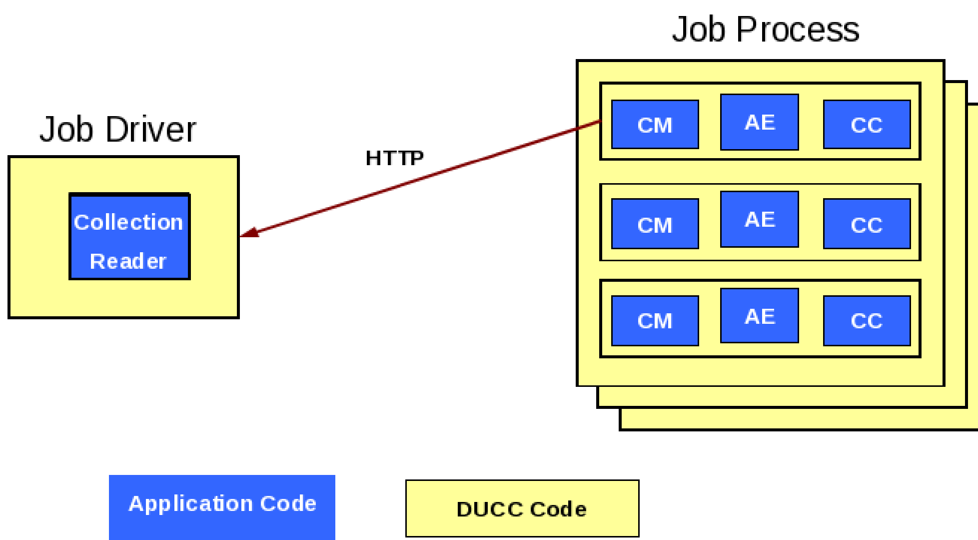

Figure 1.3: UIMA Pipeline As Automatically Scaled Out By DUCC

UIMA Pipeline with User-Supplied DD Scaled By DUCC Application programmers may supply their own Deployment Descriptors to control intra-process threading and scale-out. If a DD is specified in the job parameters, DUCC will launch each JP with the specified UIMA-AS service instantiated in-process, as depicted in Figure 1.4 below. In this case the user can still specify how many work items to deliver to the service concurrently.

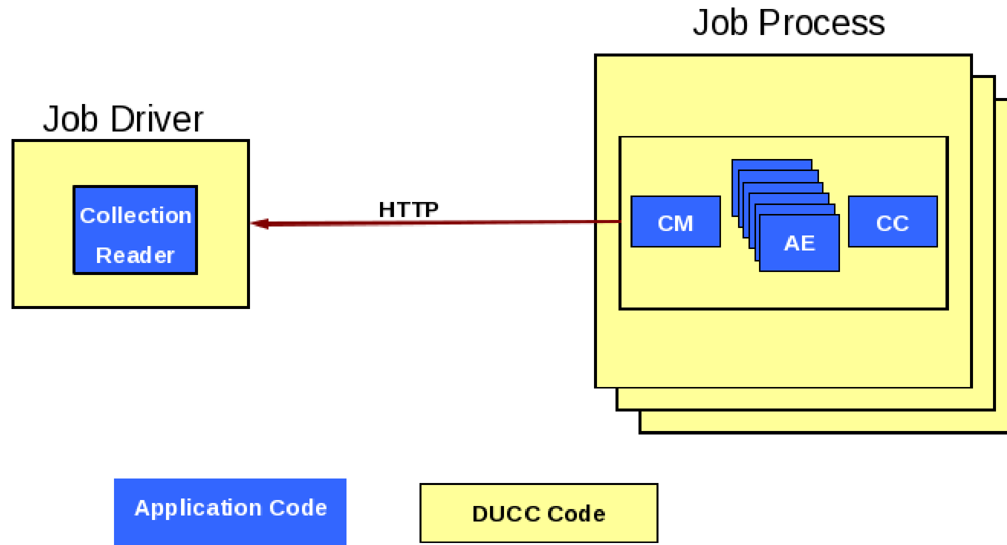

Figure 1.4: UIMA Pipeline With User-Supplied DD as Automatically Scaled Out By DUCC

1.4 Error Management

DUCC provides a number of facilities to assist error management:

- DUCC captures exceptions in the JPs and delivers them to the Job Drivers. The JD wrappers implement logic to enforce error thresholds, to identify and log errors, and to reflect job problems in the DUCC Web Server. Error thresholds are configurable both globally and on a per-job basis.
- Error and timeout thresholds are implemented for both the initialization phase of a pipeline and the execution phase.
- Retry-after-error is supported: if a process has a failure on some CAS after initialization is successful, the process is terminated and all affected CASs are retried, up to some configurable threshold.
- To avoid disrupting existing workloads by a job that will fail to run, DUCC ensures that JD and JP processes can successfully initialize before fully scaling out a job.
- Various error conditions encountered while a job is running will prevent a problematic job from continuing scale out, and can result in termination of the job.

1.5 Cluster and Job Management

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

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

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

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

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

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

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

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

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

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

Cluster Management Support DUCC provides system management support to:

- Start, stop, and query full DUCC systems.
- Start, stop, and quiesce individual DUCC components.
- Add and delete nodes from the DUCC system.
- Discover DUCC processes (e.g. after partial failures).
- Find and kill errant job processes belonging to individual users.
- Monitor and display inter-DUCC messages.

1.6 Security Measures

The following DUCC security measures are provided:

user credentials DUCC instantiates user processes using a setuid root executable named ducc_ling. See more at *ducc_ling*.

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

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

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

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

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

ActiveMQ DUCC uses ActiveMQ for administrative communication. AMQ authentication is used to prevent arbitrary processes from participating.

1.6.1 ducc_ling

ducc_ling contains the following functions, which the security-conscious may verify by examining the source in $DUCC_HOME/duccling. All sensitive operations are performed only AFTER switching userids, to prevent unauthorized root access to the system.

- Changes it's real and effective userid to that of the user invoking the job.
- Optionally redirects its stdout and stderr to the DUCC log for the current job.
- Optionally redirects its stdio to a port set by the CLI, when a job is submitted.
- "Nice"s itself to a "worse" priority than the default, to reduce the chances that a runaway DUCC job could monopolize a system.
- Optionally sets user limits.
- Prints the effective limits for a job to both the user's log, and the DUCC agent's log.
- Changes to the user's working directory, as specified by the job.
- Optionally establishes LD_LIBRARY_PATH for the job from the environment variable `DUCC_LD_LIBRARY_PATH` if set in the DUCC job specification. (Secure Linux systems will prevent LD_LIBRARY_PATH from being set by a program with root authority, so this is done AFTER changing userids).
- ONLY user *ducc* may use the ducc_ling program in a privileged way. Ducc_ling contains checks to prevent even user *root* from using it for privileged operations.

1.7 Security Issues

The following DUCC security issues should be considered:

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

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

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

Chapter 2

Glossary

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

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

DUCC Distributed UIMA Cluster Computing.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Part II

Ducc Users Guide

Chapter 3

Command Line Interface

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

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

```
description                      Test job 1

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

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

driver_jvm_args                  -Xmx500M

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

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

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

The DUCC CLI parameters are now described in detail.

3.1 The DUCC Job Descriptor

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

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

3.2 Operating System Limit Support

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

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

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

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

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

3.3 Command Line Forms

The Command Line Interface is provided in several forms:

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

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

3.4 DUCC Commands

The following commands are provided:

ducc_submit Submit a job for execution.

ducc_cancel Cancel a job in progress.

ducc_reserve Request a reservation of a full machine.

ducc_unreserve Cancel a reservation.

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

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

ducc_process_cancel Cancel an arbitrary process.

ducc_services Register, unregister, start, stop, modify, disable, enable, ignore references, observe references, and query a service.

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

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

The next section describes these commands in detail.

3.5 ducc_submit

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

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

Usage:

Script wrapper $DUCC_HOME/bin/ducc_submit *options*

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

Options:

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

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

−−**cancel_on_interrupt** If specified, the job is monitored and will be canceled if the submit command is interrupted, e.g. with CTRL-C. This option always implies −−wait_for_completion.

−−**classpath [path-string]** The CLASSPATH used for the job. If specified, this is used for both the Job Driver and each Job Process. If not specified, the CLASSPATH of the process invoking this request is used.

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

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

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

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

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

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

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

−−**driver_exception_handler_arguments [argument-string]** This is a string containing arguments for the exception handler. The contents of the string is entirely a function of the specified handler. If not specified, a *null* is passed in.

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

The built-in default exception handler supports an argument string of the following form (with NO embedded blanks):

```
max_job_errors=15
```

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

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

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

−−**environment [env vars]** Blank-delimited list of environment variables and variable assignments. Entries will be copied from the user's environment if just the variable name is specified, optionally with a final '*' for those with the same prefix. If specified, this is used for all DUCC processes in the job. Example:

```
--environment TERM=xterm DISPLAY=:1.0 LANG UIMA_*
```

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

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

in the global DUCC configuration ducc.properties.

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

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

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

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

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

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

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

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

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

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

```
--process_deployments_max 66
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
--process_failures_limit 23
```

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

```
--process_initialization_failures_cap 62
```

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

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

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

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

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

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

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

```
--process_per_item_time_max 60
```

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

```
--process_thread_count 7
```

−−**scheduling_class [classname]** This specifies the name of the scheduling class the RM will use to determine the resource allocation for each process. The names of the classes are installation dependent. If not specified, the FAIR_SHARE default is taken from the site class definitions file described here. Example:

```
--scheduling_class normal
```

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

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

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

For example,

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

where job.props contains:

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

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

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

−−**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 monitors the job and prints periodic state and progress information to the console. When the job completes, the monitor is terminated and the submit command returns. If the command is interrupted, e.g. with CTRL-C, the job will not be canceled unless −−cancel_on_interrupt is also specified.

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

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

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

3.6 ducc_cancel

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

Usage:

Script wrapper $DUCC_HOME/bin/ducc_cancel *options*

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

Options:

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

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

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

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

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

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

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

Notes: None.

3.7 ducc_reserve

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

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

Usage:

Script wrapper $DUCC_HOME/bin/ducc_reserve *options*

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

Options:

−−**cancel_on_interrupt** If specified, the request is monitored and will be canceled if the reserve command is interrupted, e.g. with CTRL-C. This option always implies −−wait_for_completion.

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

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

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

−−**memory_size [integer]** This specifies the amount of memory the reserved machine must support. After rounding up it must match the total usable memory on the machine. (Required)

−−scheduling_class [classname] This specifies the name of the scheduling class the RM will use to determine the resource allocation for each process. It must be one implementing the RESERVE policy. If not specified, the RESERVE default is taken from the site class definitions file described here.

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

−−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 By default, the reserve command monitors the request and prints periodic state and progress information to the console. When the reservation completes, the monitor is terminated and the reserve command returns. If the command is interrupted, e.g. with CTRL-C, the request will not be canceled unless −−cancel_on_interrupt is also specified. If this option is disabled by specifying a value of "false", the command returns as soon as the request has been submitted.

Notes: Reservations must be for full machines, in a job class implementing the RESERVE scheduling policy. The default DUCC distribution configures class *reserve* for full machine reservations. If there is no available machine in that class matching the requested size (after rounding up) the request is queued. The user may cancel the request with *ducc_unreserve* or with CTRL-C if −−cancel_on_interrupt was specified.

3.8 ducc_unreserve

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

Usage:

Script wrapper $DUCC_HOME/bin/ducc_unreserve *options*

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

Options:

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

−−id [jobid] The ID is the id of the reservation to cancel. (Required)

−−help Prints the usage text to the console.

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

Notes: None.

2y

3.9 ducc_process_submit

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

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

Usage:

Script wrapper $DUCC_HOME/bin/ducc_process_submit *options*

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

Options:

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

−−**cancel_on_interrupt** If specified, the remote process is monitored and will be canceled if the submit command is interrupted, e.g. with CTRL-C. This option always implies −−wait_for_completion.

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

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

−−**environment** [**env vars**] Blank-delimited list of environment variables and variable assignments. Entries will be copied from the user's environment if just the variable name is specified, optionally with a final '*' for those with the same prefix. If specified, this is used for all DUCC processes in the job. Example:

```
--environment TERM=xterm DISPLAY=:1.0 LANG UIMA_*
```

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

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

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

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

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

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

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

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

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

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

−−**scheduling_class** [**classname**] This specifies the name of the scheduling class the RM will use to determine the resource allocation for each process. The names of the classes are installation dependent. If not specified, the FIXED_SHARE default is taken from the site class definitions file described here.

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

For example,

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

where job.props contains:

```
working_directory   = /home/bob/projects
environment         = AE_INIT_TIME=10000 LD_LIBRARY_PATH=/a/bogus/path
log_directory       = /home/bob/ducc/logs/
description         = Simple Process
scheduling_class    = fixed
process_memory_size = 15
```

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

−−**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 monitors the remote process and prints periodic state and progress information to the console. When the process completes, the monitor is terminated and the submit command returns. If the command is interrupted, e.g. with CTRL-C, the request will not be canceled unless −−cancel_on_interrupt is also specified.

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

Notes:

3.10 ducc_process_cancel

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

Usage:

Script wrapper $DUCC_HOME/bin/ducc_process_cancel *options*

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

Options:

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

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

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

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

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

Notes: None.

3.11 ducc_services

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

The functions include:

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

The registration consists primarily of a service specification, similar to a job specification. This specification is used when the Service Manager needs to start a service instance. The registered properties for a service are made available for viewing from the DUCC Web Server's service details page.

Unregister This unregisters a service with the Service Manager. When a service is unregistered DUCC stops the service instance and moves the specification to history.

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

Stop The stop function stops some or all service instances.

Modify The modify function allows most aspects of a registered service to be updated without re-registering the service. Where feasible the modification takes place immediately; otherwise the service must be stopped and restarted.

Disable This prevents additional instances of a service from being spawned. Existing instances are not affected.

Enable This reverses the effect of a manual *disable* command or an automatic disable of the service due to excessive errors.

Ignore References A reference started service no longer exits after the last work referencing the service exits. It remains running until a manual stop is performed.

Observe References A manually started service is made to behave like a reference-started service and will terminate after the last work referencing the service has exited (plus the configured linger time).

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

Usage:

Script wrapper $DUCC_HOME/bin/ducc_services *options*

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

The ducc_services CLI requires one of the verbs "register", "unregister", "start", "stop", "query", or "modify". Other arguments are determined by the verb as described below.

Options:

3.11.1 Common Options

These options are common to all of the service verbs:

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

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

3.11.2 ducc_services –register [specification file] [options]

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

The *specification file* is optional. If designated, it is a Java properties file containing other registration options, minus the leading "–". If both a specification file and command-line options are designated, the command-line options override those in the specification.

The options describing the service include:

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

−−**classpath [path-string]** The CLASSPATH used for the service, if the service is a UIMA-AS services. If not specified, the CLASSPATH of the process invoking this request is used.

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

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

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

−−**environment [env vars]** Blank-delimited list of environment variables and variable assignments for the service. Entries will be copied from the user's environment if just the variable name is specified, optionally with a final '*' for those with the same prefix. Example:

```
--environment TERM=xterm DISPLAY=:1.0 LANG UIMA_*
```

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

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

in the global DUCC configuration ducc.properties.

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

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

−−**instances [n]** This specifies the number of instances to start when the service is started. If not specified, the default is 1. Each instance has the DUCC_SERVICE_INSTANCE environment variable set to a unique sequence number, starting from 0. If an instabce is restarted it will be assigned the same number.

−−**instance_failures_window [time-in-minutes]** This specifies the time in minutes that service instance failures are tracked. If there are more service instance failures within this time period than are allowed by −−*instance_failures_limit* the service's *autostart* flag is set to *false* and the Service Manager no longer starts instances for the service. The instance failures may be reset by resetting the autostart flag with the −−*modify* option, or if no subsequent failures occur within the window.

This option pertains only to failures which occur after the service is initialized.

This value is managed the a services ping/monitor. Thus if it is dynamnically changed with the −−*modify* option it takes effect immediately.

−−instance_failures_limit [number of allowable failures] This specifies the maximum number of service failures which may occur with the time specified by −−*instance_failures_window* before the Service Manager disables the service's *autostart* flag. The accounting of failures may be reset by resetting the autostart flag with the −−*modify* option or if no subsequent failures occur within the time window.

This option pertains only to failures which occur after the service is initialized.

This value is managed the a services ping/monitor. Thus if it is dynamnically changed with the −−*modify* option the current failure counter is reset and the new value takes effect immediately.

−−instance_init_failures_limit [number of allowable failures] This specifies the number of consecutive failures allowed while a service is in initialization state. If the maximum is reached, the service's *autostart* flag is turned off. The accounting may be reset by reeenabling *autostart*, or if a successful initialization occurs.

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

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

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

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

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

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

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

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

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

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

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

−−process_debug [host:port] The specifies a debug port that a service instance connects to when it is started. If specified, only a single service instance is started by the Service Manager regardless of the number of instances specified. The service instance's JVM options are enhanced so the service instance starts in debug mode with the correct call-back host and port. The host and port are used for the callback.

To disable debugging, user the −−*modify* service option to set the host:port to the string "off".

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

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

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

−−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 FIXED_SHARE default is taken from the site class definitions file described here.

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

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

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

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

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

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

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

```
service_ping_arguments=broker-jmx-port=pppp,meta-timeout=tttt
```

The keywords in the string have the following meaning:

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

Sometimes it is necessary to disable the gathering of ActiveMQ statistics through JMX; for example, if the queue is accessed via HTTP instead of TCP. To disable JMX statistics, specify the port as "none".

```
service_ping_arguments=broker-jmx-port=none
```

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

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

This parameter is required for CUSTOM services.

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

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

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

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

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

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

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

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

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

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

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

3.11.3 ducc_services –start options

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

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

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

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

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

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

is issued, five more service instances are started for service 33 for a total of ten, regardless of the number specified in the registration.

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

3.11.4 ducc_services –stop options

The stop function instructs DUCC to stop some number of service instances. If no specific number is specified, all instances are stopped.

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

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

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

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

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

Example:

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

3.11.5 ducc_services –enable options

The enable function removes the *disabled* flag and allows a service to resume spawning new instances according to its management policy.

−−enable [**service-id or endpoint**] Removes the *disabled* status, if any. Example:

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

3.11.6 ducc_services –disable options

The disable function prevents the service from starting new instances. Existing instances are not affected. Use the *ducc_services –enable* command to reset.

−−disable [**service-id or endpoint**] sets the *disabled* status. Example:

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

3.11.7 ducc_services –observe_references options

If the service is not autostarted and has active instances, this instructs the Service Manager to track references to the service, and when the last referencing service exits, stop all instances. The registered *linger* time is observed after the last reference exits before stopping the service. See the management policy section for more information.

−−observe_references [**service-id or endpoint**] Instructs the SM to manage the service as a *reference-started* service. Example:

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

3.11.8 ducc_services –ignore_references options

If the service is manually started and has active instances, this instructs the Service Manager to NOT stop the service when the last referencing job has exited. It transforms a *manually-started* service into a *reference-started* service. See the management policy section for more information.

−−ignore_references [**service-id or endpoint**] Instructs the SM to manage the service as a *reference-started* service. Example:

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

3.11.9 ducc_services –modify options

The modify function dynamically updates some of the attributes of a registered service. All service options as described under *−−register* other than the *service_endpoint* and *process_descriptor_DD* may be modified wihtout re-registering the service. In most cases the service will need to be stopped and restarted for the update to apply.

The modify option is of the following form:

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

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

The following modifications take place immediately without the need to restart the service:

- instances
- autostart

- service_linger
- process_debug
- instance_init_failures_limit

Modifying the following registration options causes the service pinger to be stopped and started, without affecting any of the service instances themselves. The pinger is restarted even if the modification value is the same as the old value. (A good way to restart a possibly errant pinger is to modify it's *service_ping_dolog* from "true" to "true" or from "false" to "false".)

- service_ping_arguments
- service_ping_class
- service_ping_classpath
- service_ping_jvmargs
- service_ping_timeout
- service_ping_dolog

3.11.10 ducc_services –query options

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

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

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

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

Below is a sample service query.

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

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

Notes:

3.12 viaducc and java_viaducc

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

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

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

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

Usage:

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

or

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

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

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

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

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

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

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

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

Chapter 4

The DUCC Public API

4.1 Overview Of The DUCC API

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

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

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

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

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

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

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

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

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

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

4.2 Compiling and Running With the DUCC API

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

4.3 Java API

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

Chapter 5

Service Management

5.1 Overview.

A DUCC service is defined by the following two criteria:

- A service is one or more long-running processes that await requests and return something in response.
- A service that is managed by DUCC is accompanied by a small program called a "pinger" that the DUCC Service Manager uses to gauge the availability and health of the service. This pinger must always be be present. DUCC will supply a default pinger for UIMA-AS services if none is specified.

  Users may supply their own "pingers" by supplying a Java class that implements the pinger API. This is referred to as a "custom" pinger in this document. There are a number of service registration options which allow specification and parametrization of custom pingers.

The pinger API enables the following functions for custom pingers:

- increase and decrease the number of service instances,
- manage failure restart policies,
- enable and disable service autostart,
- notify the Service Manager of the date of last use of a service,
- notify the Service Manager of the health and availability of a service,
- returns a string for display in the DUCC Web server to show relevant service information

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

The DUCC Service Manager implements several high-level functions:

- Ensure services are available for jobs before allowing the jobs to start.
- Enable fast-fail for jobs which reference services which are unavailable.
- Start a service when it is referenced by a job, and stop it when no longer needed.
- Optionally start a service when DUCC is booted.
- Insure services remain operational across failures.
- Report service failures.
- Run service pingers and respond to the pinger API as needed.

When work enters the system with a declared dependency on a service, one of the following actions is taken:

- If the service is not registered, the work request is automatically canceled (to avoid wasting resources on a job that is known cannot succeed.)
- If the service registered but not running, the Service Manager attempts to start it; the job remains queued until the service is started and its pinger reports good health.
- If the service exists but cannot be started, the remains queued and error status is shown in the web server. Once the service is working again the work is allowed to proceed. (Jobs already running are not directly affected, unless they also cannot access the service.)
- If the service processes are running but the pinger reports failure contacting the service, the work remains queued with error status shown in the webserver. Once the service pinger indicates the service is functional again the work is allowed to proceed.

5.2 Service Types.

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

UIMA-AS This is a normal UIMA-AS service. DUCC fully supports all aspects of UIMA-AS services with minimal effort from developers. A default pinger is supplied by DUCC for UIMA-AS services. It is legal to define a custom pinger for a UIMA-AS service.

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

DUCC also supports services that are not managed by DUCC. These are known as *ping-only* services. The registration for a ping-only service contains only keywords needed to support a pinger, which communicates with the non-DUCC service. Ping-only services must be defined as custom services; there is no default pinger provided for ping-only services.

5.3 Service Instance IDs

DUCC 2.0.0 introduces support for constant service instance IDs. As a service is being started, the SM assigns monotonically increasing IDs to each service instance, starting with ID 0, up the the maximum number of instances started.

If an instance exits unexpectedly, the SM re-spawns it (unless a failure threshold has been exceeded). The new instance is assigned the same instance ID as the instance it replaces. This insures that, for example, instance "three" is always started as instance "three", maintained constant over failures and SM restarts.

The instance ID is communicated to the process through the environment with the key `DUCC_SERVICE_INSTANCE`. This key may also be used in service registrations if it is desired to pass the instance ID via parameters of some sort. For example:

```
service_jvm_args   -DSERVICE_ID=${DUCC_SERVICE_INSTANCE}
process_executable_args -i ${DUCC_SERVICE_INSTANCE}
```

5.4 Service References and Endpoints

Services are identified by an entity called a *service endpoint*. Jobs and other services use the registered service endpoint to indicate dependencies on specific services.

A service endpoint is of the form

```
<service-type>:<unique id>
```

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

The *unique id* is any string needed to ensure the service is uniquely named. For UIMA-AS services, the unique ID must be the same as the service endpoint specified in service's DD XML descriptor. The UIMA-AS service endpoint is always of the form:

```
queue-name:broker-url
```

where *queue-name* is the name of the ActiveMQ queue used by the service, and *broker-url* is the ActiveMQ broker URL. Sample DUCC Service endpoints:

```
UIMA-AS:WikipediaSearchServices:tcp://broker1:61616
UIMA-AS:GoogleSearchServices:http://broker2:61618
```

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

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

5.5 Service Management Policies

The Service Manager implements these policies for managing services:

Autostarted Services An autostarted service is automatically started when the DUCC system is first booted. If an instance should die, DUCC automatically restarts the instance and continually maintains the registered number of service instances.

By default, to handle fatal errors in *autostarted* services, The Service Manager maintains a time window in which only a specific number of instance failures may occur. If the number of failures within that window of time is excessive DUCC will set a *disabled* flag and no longer restart instances. Instance which do not fail are left running. The *disabled* flag must be manually reset once the problem is resolved before new instances can be started.

The default failure policy is implemented in the service pinger.. Service owners may redefine the default policy by supplying their own pingers for a service.

Reference-started Services A reference-started service is a registered service that is started only when referenced by another job or service. If the service is already started, the dependent job/service is marked "Services Available" and can be scheduled.. If not, the service registry is checked and if a matching enabled service is found, it is started by DUCC. While the service is being started, jobs are held "Waiting For Services" to ensure the service is viable. Once the service has completed initialization and the pinger indicates it is viable, all work waiting on it is then marked "Services Available" and started.

To handle fatal errors in *reference-started* services, The Service Manager maintains a time window in which only a specific number of instance failures may occur. If the number of failures within that window of time is excessive DUCC will set a *disabled* flag and no longer restart instances. Instance which do not fail are left running. The *disabled* flag must be manually reset once the problem is resolved before new instances can be started. This default policy may be overridden by custom pingers.

When the last job or service that references the on-demand service exits, a timer is established to keep the service alive for a while, in anticipation that it will be needed again soon. When the keep-alive timer expires, and there are no more dependent jobs or services, the reference-started service is automatically stopped to free up its resources for other work. The time the service is allowed to remain alive is known as its *linger* time and can be controlled with the *service_linger* keyword in the service registration.

Manually started services A service may be started via the CLI if it is not already running and in the absence of references by other work. A service which is manually stared by the CLI can only be stopped manually by the CLI.

As is the case for *autostarted* and *reference-started* services, failed instances will be restarted unless the number of failures within the failure window is exceeded and the *disable* flag is set.

Ping-Only Services Ping-only services consist of only a ping thread. The service itself is not managed in any way by DUCC. This is useful for managing dependencies on services that are not under DUCC control: the pinger is used to assess the viability of the external service and prevent dependent jobs from continuing if the service is unavailable.

Only CUSTOM services may be defined as ping-only services in this version of DUCC.

Dynamically Changing Service Policies A service may be *stopped*; that is, no instances are running. This state can occur if the service has experienced too many errors within its failure window, in which case the service is *disabled*, or because the service is not *autostarted* or *referenced* by other work.

If a manual *stop* is issued the service will be automatically *disabled* to insure it cannot be restarted (by *reference* or at boot with *autostart*) without manual intervention.

In all cases, if a service is *disabled*, it must be manually *enabled* using the CLI.

It is possible, via the CLI, to dynamically switch any service from any management policy to any other policy, as shown in the following table.

See the *Service CLI* reference for details on the various commands described in this section.

Current Mode	Desired Mode	Action	Notes
Autostart	Manual	Use CLI to modify registration to *autstart false*.	Service does not stop until requested by CLI. Service will not start at DUCC boot.
Autostart	Reference	Use CLI to modify registration to *autostart false* and *observe references*.	Service stops after last reference exits, plus *linger* time.
Autostart	*Stopped*	Use CLI to stop the service.	The CLI stop will by necessity *disable* the service to insure it remains stopped.
Reference	Autostart	Use CLI to modify registration to *autostart true*.	Service continues to run after last reference exits. Service always started at DUCC boot.
Reference	Manual	Use CLI to *ignore references*.	Service continues to run after last reference exits.
Reference	*Stopped*	Use CLI to stop the service.	The CLI stop will by necessity *disable* the service to insure it remains stopped.
Manual	Autostart	Use CLI to modify registraiton to *autostart true*.	Service will be started on DUCC boots.
Manual	Reference	Use CLI to *observe references*.	Service will stop after last referencing job exits, plus *linger* time..
Manual	*Stopped*	Use CLI to stop the service.	The CLI stop will by necessity *disable* the service to insure it remains stopped.
Stopped	Autostart	Use CLI to modify registration to *autostart true*.	Service will start immediately. It may be necessary to *enable* the service as well.
Stopped	Reference	Submit a job or service that references the service.	It may be necessary to *enable* the service as well. The service will stop after the last referencing work exits, plus *linger*.
Stopped	Manual	Use CLI to start the service.	The CLI start will also *enable* the service if necessary.

5.6 Service Pingers

A service pinger is a small program that queries a service on behalf of the DUCC Service Manager. A default pinger is provided for UIMA-AS services and provides the following functions:

- Determine if the service is responsive by issuing a UIMA-AS "get-meta" call to the service.
- Determine the health of the service by issuing a JMX call to the UIMA-AS broker to collect queueing statistics.
- Manage the failure window of the service.
- Returns a string with basic ActiveMQ statistics about the service, or error information if the service is deemed unusable.
- Returns date of last use of the service (as determined by presence or absence of service producers attached to the service queue).

Users may supply their own pingers. The following additional functions are available for pingers. Note that a *custom* pinger MAY be supplied for UIMA-AS services, and MUST be supplied for CUSTOM services. Custom pingers use the Service Manager's "pinger" API to perform the following tasks:

- Inform the Service Manager if the service is viable.
- Inform the Service Manager if the service is "healthy". Service "health" is a heuristic used in the DUCC Web server as an alert that a service is responding but may not be performing well.
- Manage service failure policies. Default failure-window policy is provided to all pingers by the DUCC API handler (optional).
- Return a string describing current service status, for use by the web server.
- Instruct the service manager to increase the number of instances (optional).
- Instruct the service manager to decrease the number of instances (optional).
- Enable and disable the services autostart flag (optional).
- Enable logging of a service's health and state (optional).
- Return date of last-use to the Service Manager for display in the webserver (optional).

5.6.1 The Pinger API

Pingers are passed static information about the service at pinger-initialization time, and subsequently, current state of the service is provided on each call (ping).

Information provided at initialization follows. Most of this is provided in fields in the *AServicePing* base class. See the Javadoc for specific field names and types.

Pinger Initialization Data

Data provided once, during pinger initialization, includes:

Arguments This is the *service_ping_arguments* string from the service registration.

Endpoint This is the CUSTOM:string or UIMA-AS:string endpoint provided in the service registration.

Monitor Rate This is the rate at which the pinger will be called by the SM, as provided in DUCC's configuration.

Service ID This is the unique numeric service ID assigned to the service by DUCC.

Log Enabled Whether the service log is enabled, as specified by the *service_ping_dolog* registration parameter.

Maximum Allowed Failures This is the value of the *instance_failures_limit* parameter, provided by DUCC configuration and optionally overridden by the service registration.

Instance Failure Window This is the value of the *instance_failures_window* parameter, provided by DUCC configuration and optionally overridden by the service registration.

Autostart Enabled This indicates whether the service registration currently has the *autostart* flag enabled.

Last Use This is the time of last known use of the service, persisted and maintained over SM restarts. It is 0 if unknown or the service has never been used.

Pinger Dynamic Data

Dynamic information provided to the pinger in each call (ping) consists of:

All Instance Information This is an array consisting of the unique integer IDS of all running processes implementing the service. This includes instances which may not be currently viable for some reason (still initializing, for example).

Active Instance Information This is an array consisting of the unique integer IDS of all running processes implementing the service. This is a subset of "All Instance Information" and includes only the service instances that are advanced to Running state.

Reference Information This is an array consisting of the unique integer IDS of all DUCC work (Jobs, other Services, etc) currently referencing the service.

Autostart Enabled The current state of the service's autostart flag.

Run Failures This is the total number of instance failures for the service since the last start of the SM.

Only a Java API is supported.

5.6.2 Declaring a Pinger in A Service

The following registration options are used for declaring and configuring pingers. Any of these may be dynamically modified with the service CLI's −−*modify* option. Dynamically changing these causes the current pinger to be terminated and restarted with the new configuration. See ducc_services for details of the options:

- service_ping_arguments
- service_ping_class
- service_ping_classpath
- service_ping_jvmargs
- service_ping_timeout
- service_ping_dolog
- instance_failures_window
- instance_failures_limit

5.6.3 Implementing a Pinger

Pingers must implement the class `org.apache.uima.ducc.cli.AServicePing`. See the Javadoc for the details of this class.

Below is a sample CUSTOM pinger for a hypothetical service that returns four integers in response to a ping. It illustrates simple use of the three required methods, *init()*, *stop()*, and *getStatistics()*.

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

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

    public void stop()  {  }

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

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

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

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

Figure 5.1: Sample UIMA-AS Service Pinger

5.6.4 Building And Testing Your Pinger

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

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

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

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

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

```
service_ping_dolog = true
```

This will log any output from `System.out.println()` to the declared log directory for the service. If not specified in the reqistration, this directory is:

```
$HOME/ducc/logs/S-<serviceid>/services
```

where <`servicid`> is the DUCC-assigned ID of your service.

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

```
service_ping_dolog = false
```

A sample service registration may look something like the following. Note that you do not need to include any of the DUCC jars in the classpath for the pinger. DUCC will add the jars it requires to interact with the pinger automatically. (However you may need other jars to provide UIMA, UIMA-AS, ActiveMQ, Spring, or other function.)

```
bash-3.2$ cat myping.svc

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

3. Register and start the service and pinger Start up your custom service so the pinger with the registration containing lines similar to those above. As soon as the service instance is in DUCC state *Running* the SM starts the pinger.

Check the web server to make sure the service "comes alive". Check your pinger's debugging log if it doesn't. Once registered, you can dynamically modify and restart the pinger at any time without re-registering the service or restarting the service by use of the `--modify` option of the *ducc_services CLI:*

```
ducc_services --modify <serviceid> --service_ping_dolog true
ducc_services --modify <serviceid> --service_ping_class OtherCustsomPing
                                   --service_ping_classpath /myhome/OtherCustomPing.class
```

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

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

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

5.6.5 Globally Registered Pingers

A user-built pinger may be registered with DUCC so that it can be globally used by any DUCC service. To do this, a registration file containing only pinger-specific parameters is created in DUCC's run-time directory. Such a pinger may then be designated for a service by using its registered filename instead of its class in the *service_ping_class* field of a registration. There is no API or CLI to register such a pinger; only a DUCC administrator may create a global ping registration.

A globally-registered pinger may then be designated to run as a thread inside the SM or as a process spawned and managed by the SM. A pinger that runs in a thread in the SM is called an *internal* pinger, and one that runs in a process is called an *external* pinger. An *internal* pinger generally has nearly unmeasurable impact on the system, whereas *external* pingers will occupy full JVMs with processes of 50-100MB or more.

A service may override any of the options of a globally-registered *external* pinger, thus allowing significant reuse of existing code. Only the *service_ping_arguments* of an *internal* pinger may be overridden however.

The default UIMA-AS pinger is permanently registered as an *internal* pinger.

Globally registered pingers use a special boolean property, not supported by the *ducc_services* API/CLI, "internal", to determine whether the pinger is to be run internally to SM or as an external process. Only the DUCC administrator may update a global pinger's registration to "internal", to insure such pingers are properly vetted and approved by the installation.

More Details of registering global pingers is found in the *Administration section* of this document.

5.7 Sample Pinger

A sample custom UIMA-AS pinger is provided in the Examples directory shipped with DUCC in

```
DUCC_HOME/examples/src/org/apache/uima/ducc/ping
```

This pinger increases or decreases the number of service instances based on the queue statistics found by querying ActiveMQ. The goal of this pinger is to maintain the ActiveMQ "enqueued time" to be no more than some multiple of the average service time for a single item. The factor used is a parameter passed in with the argument string.

5.7.1 Using the Sample Pinger

The following arguments may be specified to use the sample pinger with any UIMA-AS service. The *service_ping_arguments* are specific to this pinger.

```
service_ping_class=org.apache.uima.ducc.ping.SamplePing
service_ping_arguments=meta-timeout=15010,broker-jmx-port=1099,window=5,min=1,
                       max=20,max-growth=3,fast-shrink=true,goal=2.5
service_ping_classpath = ${DUCC_HOME}/lib/uima-ducc/examples/*:
                         ${DUCC_HOME}/apache-uima/lib/*:
                         ${DUCC_HOME}/apache-uima/apache-activemq/lib/*:
                         ${DUCC_HOME}/lib//springframework/*
service_ping_dolog=True
service_ping_timeout=10000

instance_failures_window = ${ducc.sm.instance.failure.window}
instance_failures_limit  = ${ducc.sm.instance.failure.max}
```

The full source for the sample pinger is found in

```
DUCC_HOME/examples/src/org/apache/uima/ducc/ping/SamplePing.java
```

The following arguments are accepted by this pinger and may be specified in a single single comma-delimited string containing the following initialization parameters:

meta-timeout Defines how long to wait for *get_meta* to return.

broker-jmx-port Defines the JMX port of the service's broker.

window Defines the shrinkage/growth window size, in minutes.

enable-log Enable extra logging.

min The minimum number of service instances to maintain.

max The maximum number of service instances to allow.

max-growth The maximum number of instances to grow in a single request.

fast-shrink If set, allow services to shrink if the queue depth is 0, even if consumer are connected. Otherwise we do not shrink if consumers are attached to the queue.

goal The multiplier of the ActiveMQ Broker's *average enqeue* time to attempt to maintain by managing the number of instances.

5.7.2 Understanding Sample Pinger

The best way to understand this pinger is to examine the code itself in the Examples directory. Here we provide a brief line-by-line synopsis of the code.

void init(String args, String ep) This required method examines the service arguments and endpoint and establishes a monitor to issue *get-meta* calls to the service and *JMS* calls to the ActiveMq broker. The argument string *args* is described above. The endpoint *ep* is the service endpoint used to register the service.

Lines 100-119 These lines parse the endpoint *ep* its components comprising the UIMA-AS queue name and the URL to the service broker.

Lines 121-125 These lines disable most UIMA-AS logging as these messages can be quite numerous. However, during debugging it may be desired to change the logging levels here.

Lines 130-172 These lines parse the service argument string *args* into its constituent parts and places the values in variables. They initialize the expansion and deletion window and normalize it to one slot per minute, regardless of the actual ping rate.

The window normalization uses the DUCC-supplied value *monitor_rate* to determine the number of slots in the windows.

Lines 176-177 These lines initialize the DUCC-supplied *UimaAsServiceMonitor* that queries the UIMA-AS queues, and it resets the queue statistics via JMX so the monitor can make accurate measurements.

Lines 181-187 These lines implement the required *stop* method which is invoked when the Service Manager needs to stop the pinger for any reason. They stop the ActiveMQ queue monitor and emit a shutdown message.

Lines 191-240 These lines define the required *getStatistics* method. This method collects ActiveMQ statistics, issue *get-meta* to the service to see if it is responding, sets the formatted information string into the ping reply, and invokes the code to calculate a potential redeployment of service instances.

Lines 245-248 These lines override the optional *getLastUse* method which simply returns the time of last known use of the service. The actual value is calculated in the pinger-specific *calculateNewDeployment* method, described below.

Lines 253-298 These lines define the pinger-specific *calculateNewDeployment* method. This is invoked after *get-meta* is called and after the UIMA-AS queue has been queried in ActiveMQ. This is the key method of this pinger. It uses information passed in on the last ping from the Service Manager in conjunction with information in the ActiveMQ queue to determine if more, or fewer service instances are needed to meet the performance goals. If fewer instances are needed, it selects specific instances to stop. The method is *described in detail* below.

Lines 407-410 These lines override the optional *getAdditions* method. The method returns the number of new service instances required to meet performance goals, as calculated in *calculateNewDeployment*.

Regardless of what this method returns, the Service Manager may choose not to start new instances, based on its configured maximum, *ducc.sm.max.instances* as defined in *ducc.properties*.

Lines 416-419 These lines override the optional *getDeletions* method. This method returns the specific service instances to be stopped, if any.

The DUCC-assigned unique IDs of all service instances are passed in to the pinger on each ping. These instances are monotonically increasing over time so pingers may assume that lower numbers represent older instances.

Lines 429-480 These lines define a class used as a call-back on the UIMA-AS *get-meta* requests to determine the host and PID of the service instance responding to the *get-meta*. If the *get-meta* request should timeout, this information can be used to help identify ailing or overloaded service instances.

5.7.3 Calculating New Deployments in the Pinger

his section details the use of ActiveMQ queue statistics in conjunction with the Service Monitor data to calculate the number of service instances to increase or decrease.

It is important that this code be very careful about "smoothing" the performance statistics to keep growth and shrinkage stable. Things to take into consideration include:

1. Immediately after a new service instance becomes available to serve, if there is demand for this service, the ActiveMQ statistics will fluctuate for a few minutes until traffic stabilizes. Thus decisions based on these statistics must reflect history as well as current information.
2. Immediately after a client begins to use a service, the statistics will also fluctuate, again requiring smoothing.
3. The DUCC work dispatching model will not over-dispatch work to the job processes. Thus actual demand on a service is a function of the number of actively deployed and initialized JPs. If the number of JPs decreases due to preemption, demand on the service by that job will decrease proportionally. Similarly, demand can increase as the job expands.

   It is common for demand on a service to ramp up slowly as a job enters the system, and increase rapidly as a job completes its initialization phase and starts to double. Thus, the ActiveMQ statistics can be quite erratic for a while, until the job stabilizes.

   This again requires some sort of smoothing of the data when making decisions about service growth and shrinkage.

To handle this data smoothing, the SamplePing classes uses two time-based *windows*, one for growth, and one for shrinkage, to keep growth and shrinkage stable. The window size is defined in the service ping argument *window*. Each window period, if more services are needed, a mark is made in the current slot of the *expansion window*;

otherwise the current slot is cleared. Similarly, each period, if fewer services are needed, a mark is made in the *shrinkage window*; otherwise, the current slot is cleared.

After the marks are made, if the *expansion window* has all slots filled, a request for new processes is made; thus, a short period of increased does not destabilize the system with a request for services that may be of little use. Additionally, when a request is made, the number of new processes requested is capped by the ping argument *max-growth* to insure that the service grows smoothly. And finally, if the service is already at some configured maximum number of instances, defined by the *max* parameter, no additional instances are requested.

Similarly, the *shrinkage window* is used to govern shrinkage. All slots must be filled, indicating the service has been over-provisioned for a while, before a request is made to delete instances. The number of instances is never reduced below the configured *min* value. As well, this particular pinger never shrinks by more than a single instance at a time, on the reasoning that it is more costly to start a new service than to maintain one for too long. Only if there is no long-term use of the extra instances are they reduced (as determined by the window).

Given this introduction, we describe the key method in detail.

Lines 262-277 These lines extract four quantities from the ActiveMQ statistics:

1. Average enqueue time, eT
2. Current queue depth, Q
3. The current number of service consumers *cc*
4. The current number of service producers *pc*

The code then gets the DUCC IDs of all the currently started service instances, and the number of instances that are started but still in their "initialization" phase. This is important because instances that are still initializing are not servicing the queue, but will soon start to do so. The current ActiveMQ statistics reflect do NOT yet reflect this however, they reflect only the instances that are actually serving.

Finally, if there are service producers, we note the time of day to return to the SM as the last known use of this service by some process.

Lines 267 This line calculates the number of Java threads per service instance, needed to calculate the maximum capacity of the service in its current deployment.

(Note that in each UIMA-AS service, UIMA-AS itself occupies one thread, used to manage the service, and this thread manifests itself as a consumer on the queue.)

Line 301 This declares *new_ni*, the number of additional instances, if any. At the end of this method, new_ni will either be 0 or >0.

Lines 303-312 If the current queue depth is 0 (Q == 0), we know a number of things:

1. The service is not over-provisioned; there is no work queued and waiting for some service. We therefor do not need to expand.
2. If there are no consumers, i.e. no clients that need work done, we are potentially over-provisioned, so we fill in a slot in the expansion window.

   If there *are* consumers, we may not want to shrink because it is possible that one of the service instances is busy; we cannot tell. So we allow the *fast-shrink* ping argument to govern whether or not connected consumers may prevent service shrinkage.

There is nothing else that can be said about a service if its current queue depth is 0.

Lines 312-360 If the queue depth is non-zero we are able to calculate the total service capacity and the amount each instance contributes to the total capacity. From this we can determine

1. whether the service is performing at or near its goal,
2. if the service is performing worse than its goal, how many new instances are needed to meet the goal, and
3. if the service is performing better than its goal, how many instances can be given up and still meet the goal.

Details follow.

Lines 314 and 315 The average time a single instance takes to serve a single request, *Ti* is given by the simple formula

```
Ti = (eT / Q) * active
```

where

eT is the average time an item stays in queue (from AMQ),

Q is the current queue depth (from AMQ),

active is the current number of service instances (from SM)

Therefore the time taken by a single thread *Ti* is given by

```
Tt = Ti * nthreads
```

Lines 319 and 320 We want *Tt* to become close to the current

```
Tt * goal
```

where *goal* is given by the ping arguments. The current ratio of actual service time to desired is then given by

```
r = eT / g
```

Because we know that the DUCC job driver will never over-commit; that is, we know the current demand will remain constant unless the jobs using the service expand or contract (which are relatively rare events), we can state that the number of service instances required is directly proportional to r.

If $r > 1$ we may need more instances to meet our *goal* and if $r < 1$ we may be over-provisioned.

Lines 325-347 If $r > 1$ we may be over-provisioned. We calculate the number of required instances by multiplying the current instances by r and rounding down. We account for instances that we know are starting but not yet started, cap on max instances per service, and again on max growth per cycle.

If we still require additions, we make a mark in the expansion window, otherwise we clear the expansion window.

Lines 349-360 If $r < 1$ we need to calculate shrinkage. Because starting instances is expensive we conservatively use $r < .5$ instead and make a mark in the shrinkage window.

Otherwise we clear the mark in the shrinkage window.

Lines 367-396 Finally we sum across the shrinkage and expansion windows. If either window is full, we schedule growth (line 375, set the variable *additions*) or shrinkage (line 388, set *deletions*).

Note that to schedule shrinkage, we must choose a specific instance. In this case we choose the *newest* instance, i.e. the one with the largest DUCC ID, as it is most likely not to have initialized, or perhaps not to have "warmed up" (i.e. caches filled, etc.). We could choose more than one but this pinger is conservative and only shrinks by one instance each time.

5.7.4 Summary of Sample Pinger

This pinger illustrates these functions over-and above the functions provided by the default UIMA-AS pinger:

1. Use of pinger-specific arguments
2. Use of information provided by SM on each ping (service instances active, total service instances,
3. Use of performance information acquired from ActiveMQ
4. Requesting new service instances of the SM
5. Requesting that instances be removed by SM,
6. Setting of last-use of a service

It illustrates one mechanism for smoothing growth and shrinkage of a service to prevent thrashing in your system.

It illustrates one mechanism for determining the actual performance of a service by analyzing ActiveMQ queueing statistics.

It illustrates the use of "globally registered pingers."

Chapter 6

Job Logs

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

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

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

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

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

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

   where

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

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

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

   where

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

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

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

`JOBID-TYPE-NODE-PROCESS.log`

where

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

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

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

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

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

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

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

In this example,

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

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

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

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

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

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

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

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

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

Chapter 7

DUCC Web Server

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

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

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

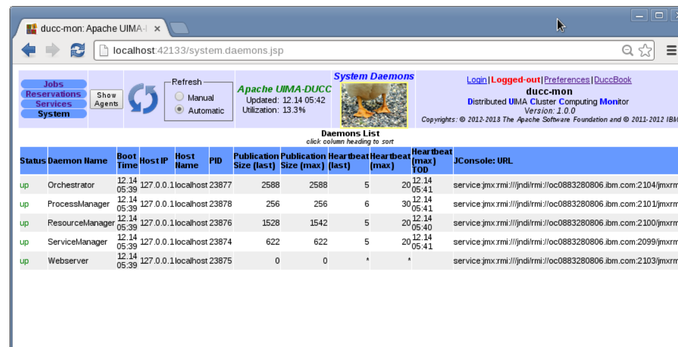

Figure 7.1: Sample Webserver Page

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

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

There is a behavior difference between refresh and reload.

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

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

Two different table styles are supported:

- Scroll, and
- Classic.

Table styles are switched using the *Preferences* link.

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

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

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

Note that not all pages have a scroll version - some only have a classic version.

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

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

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

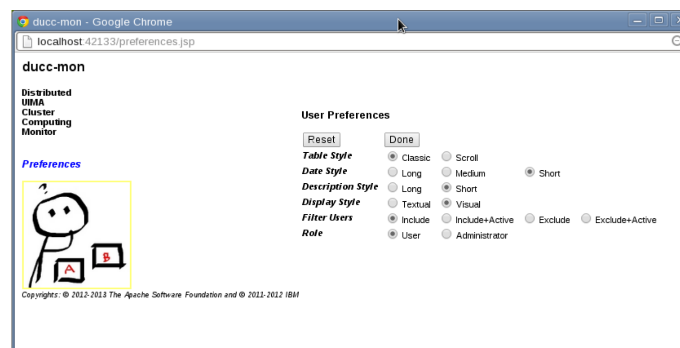

Figure 7.2: Preferences Page

7.1 Common Links

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

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

Authentication

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

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

Preferences The following preferences may be set:

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

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

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

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

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

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

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

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

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

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

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

Viz This opens a page with a visualization of the system hosts, showing all scheduled work in the system.

7.2 Jobs Page

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

The Jobs page contains the following columns:

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

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

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

User
This is the userid of the job owner.

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

State

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

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

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

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

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

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

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

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

Completed - The job is complete.

Reason or Extraordinary Status

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

EndOfJob - The job and completed ran with no errors.

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

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

CanceledBySystem - The job was canceled because DUCC was shutdown.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Size
This is the declared memory size of the job

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

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

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

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

This usually represents the quantity derived from the following formula:

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

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

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

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

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

Description

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

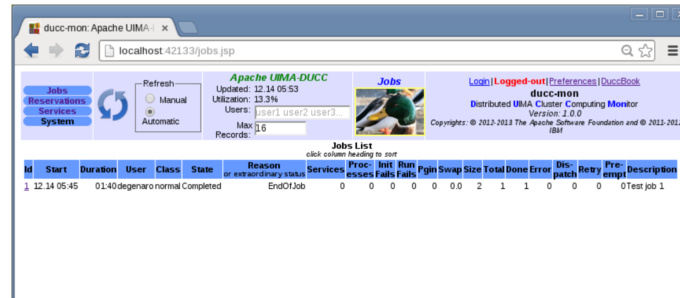

Figure 7.3: Jobs Page

7.3 Job Details Page

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

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

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

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

Specification This tab shows the job specification for the job.

7.3.1 Processes

The processes page contains the following columns:

Id

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

Log

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

Size

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

Hostname

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

PID

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

State Scheduler

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

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

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

Reason Scheduler or extraordinary status

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

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

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

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

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

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

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

JobCompleted - The process is canceled because of DUCC restart.

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

InitializationTimeout - The UIMA initialization phase exceeded the configured timeout.

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

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

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

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

State Agent

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

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

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

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

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

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

FailedInitialization The process died during the UIMA initialization phase.

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

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

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

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

Reason Agent

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

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

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

Deallocated WHAT IS THIS?

ExceededShareSize The process exceeded it's declared memory size.

ExceededSwapThreshold The process exceeded the configured swap threshold.

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

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

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

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

AdministratorInitiated The process was canceled by an administrator.

UserInitated The process was canceled by the owning user.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Figure 7.4: Processes Tab

7.3.2 Work Items

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

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

Id
This is the name of the work item.

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

ended The work item is complete.

error The work item ended with errors.

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

operating The work item is current being executed.

retry The work item is being retried.

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

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

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

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

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

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

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

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

Figure 7.5: Work Items Tab

7.3.3 Performance

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

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

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

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

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

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

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

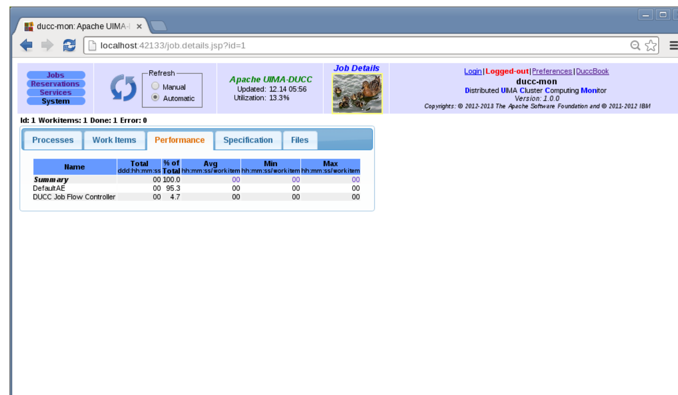

Figure 7.6: Performance Tab

7.3.4 Specification

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

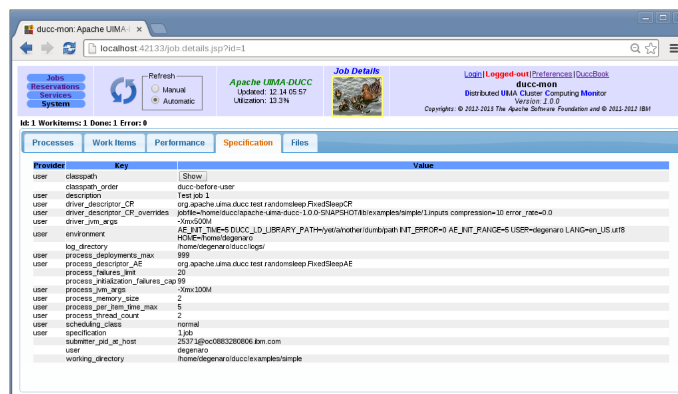

Figure 7.7: Specification Tab

7.4 Reservation Page

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

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

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

The Reservations page contains the following columns:

Id

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

Start
This is the time the reservation was mode.

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

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

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

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

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

Assigned - The reservation is active.

Completed - The reservation has been terminated.

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

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

Reason

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

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

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

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

CanceledByAdmin - The DUCC administrator released the reservation.

CanceledByUser - The reservation owner released the reservation.

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

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

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

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

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

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

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

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

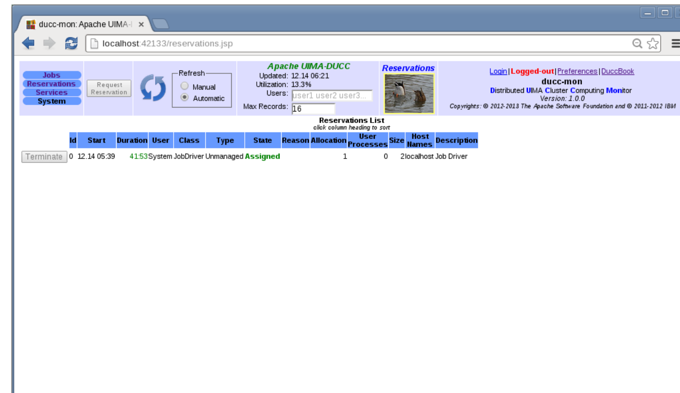

Figure 7.8: Reservations Page

7.5 Managed Reservation Details Page

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

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

Specification This tab shows the specification for the process.

7.5.1 Processes

The processes page contains the following columns:

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

```
RESID.SHAREID
```

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

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

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

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

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

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

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

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

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

Reason Scheduler or Extraordinary Status
These are the same as for the job details.

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

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

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

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

7.5.2 Specification

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

7.6 Services Page

This page shows details of all services.

The Services page contains the following columns:

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

Name
This is the unique service endpoint of the service.

Type
This is the service type, which is one of

- Registered
- Ping-Only

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

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

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

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

NotAvailable No service instance is running or initializing.

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

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

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

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

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

- *Good*
- *Poor*

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

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

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

User
This is the userid of the service owner.

Class
This is the scheduling class the service is running in.

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

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

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

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

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

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

7.7 Service Details Page

This page shows details of the processes which implement.

The information is divided between two tabs:

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

Specification This tab shows the specification for the service.

7.7.1 Processes

The processes page contains the following columns:

ID

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

```
RESID.SHAREID
```

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

Log

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

Size

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

Hostname

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

PID

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

State Scheduler

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

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

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

Reason Scheduler or Extraordinary Status

These are the same as for the job details.

State Agent

These are the same as for the job details.

Reason Agent

These are the same as for the job details.

Time Init

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

Time Run

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

Time GC

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

Pgin

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

Swap

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

%CPU

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

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

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

7.7.2 Specification

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

The specification for a Service contains two types of entries:

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

7.8 System Details Page

This page shows information relating to the DUCC System itself:

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

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

Daemons This shows the status of all DUCC processes.

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

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

7.8.1 Administration

This page has two tabs:

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

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

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

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

7.8.2 Classes

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

7.8.3 Daemons

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

The columns shown on this page include

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

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

Daemon Name
This is the name of the process.

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

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

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

PID
This is the Unix processid of the DUCC process.

Publication Size (last)
This shows the size of the most recent state publication of the process, in bytes.

Publication Size (max)
This shows the size of the largest state publication of the process, in bytes.

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

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

Heartbeat (max) TOD
This shows the time the longest delay of a state publication occurred.

JConsole URL
This is the jconsole URL for the process.

7.8.4 Machines

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

The columns shown on this page include

Status

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

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

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

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

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

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

IP

This is the IP address of the node.

Name

This is the hostname of the node.

Reserve(GB) size

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

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

Memory(GB) total

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

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

Swap(GB) in use

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

Alien PIDs

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

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

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

Shares (total)

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

Shares(in use)

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

Heartbeat(last)

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

7.9 Visualization

This page shows a visualization of all scheduled work. Every host is represented by a square whose area is proportional to the amount of memory on the host. If work is scheduled to a host, it is represented by a rectangle whose area is proportional to the amount of memory that is scheduled for the work. In a multi-user environment, each userid is mapped into a different color, making it possible to see the usage per-user.

Hovers are provided to show the real memory size of each host, the schedulable memory for each host, and the amount of memory scheduled for each bit of work.

If multiple allocations are made on a single host for the same job or service, the rectangles are combined into a single rectangle, reducing clutter and better showing the actual usage of the job (or service).

Clicking on any box representing scheduled work sends the browser to the details page for the corresponding work.

The screenshot below shows a visualization with a handful of 127GB hosts, 48GB hosts, and 32GB hosts. Regular UIMA-AS jobs show as untextured boxes; for example, job 6080, owned by user Hilaria, running in a 37GB allocation in host bluej291-41 which is a 127GB host.

Hosts bluej291-45 and 291-46 are running Managed Reservations, which are shown with crosshatches from lower-left to upper right.

Hosts bluej291-37 and bluej291-40 are running Unmanaged Reservations, shown with vertical-horizontal crosshatches.

Below bluej291-34, bluej291-36, bluej293-49, and bluej293-60 are running DUCC-managed services, shown by crosshatching from upper-left to lower-right.

The host representations may be sorted by clicking on the “size” or the “name” text near the top of the display.

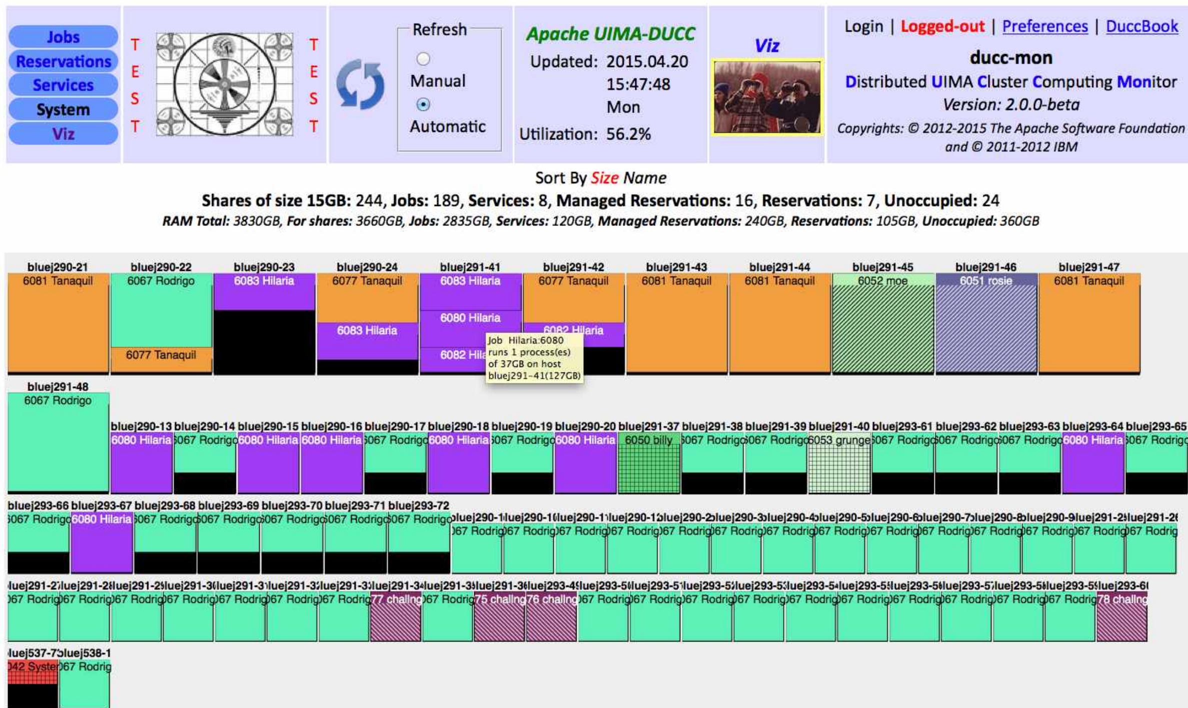

Figure 7.9: Visualization

Part III

Programming Model And Applications

Chapter 8

Building and Testing Jobs

8.1 Overview

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

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

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

8.1.1 Basic Job Process Threading Model

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

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

8.1.2 Alternate Pipeline Threading Model

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

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

8.1.3 Overriding UIMA Configuration Parameters

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

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

8.2 Collection Segmentation and Artifact Extraction

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

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

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

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

8.3 CAS Consumer Changes for DUCC

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

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

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

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

8.4 Job Development for an Existing Pipeline Design

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

To debug a Job Process with eclipse, first create a debug configuration for a "remote java application", specifying "Connection Type = Socket Listen" on some free port P. Start the debug configuration and confirm it is listening

on the specified port. Then add to the job specification process_debug=port, where port is the value P used in the running debug configuration.

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

8.5 Job Development for a New Pipeline Design

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

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

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

8.5.1 Collection Reader (CR) Characteristics

A DUCC Job CR sends Work Item CASes to the Job Processes. These CASes contain references to the data to be read by the Job Processes. Typically the CR Type System will be very small; in the DUCC sample applications the CR Type System only contains the Workitem Feature Structure described below.

Note: It is important not to include the analytic Type System in the CR. These Type Systems can be quite large and will significantly increase the size of each Work Item CAS. The Job Driver process maintains a CAS pool which must be as large as the total number of processing threads active in a job.

8.5.2 DUCC built-in Flow Controller

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

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

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

8.5.3 Workitem Feature Structure

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

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

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

inputspec (String) - reference to Work Item input data

outputspec (String) - reference to Work Item output data

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

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

bytelength (Integer) - size of Work Item

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

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

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

8.5.4 Deployment Descriptor (DD) Jobs

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

8.5.5 Debugging

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

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

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

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

Chapter 9

Sample Application: Raw Text Processing

9.1 Application Function and Design

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

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

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

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

9.2 Configuration Parameters

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

InputDirectory path to directory containing input files.

OutputDirectory path to directory for output files.

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

Encoding (optional) character encoding of the input files.

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

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

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

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

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

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

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

9.3 Set up a working directory

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

Copy to this directory the example job specification files:

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

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

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

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

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

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

9.4 Download and Install OpenNLP

Download the OpenNLP source distribution from http://opennlp.apache.org and follow the directions in the *UIMA Integration* section of the included documentation to build the UIMA pear file. Then *install* the UIMA pear file in a directory (which we will refer to below as $OPENNLP_HOME) with the *runPearInstaller* script and test it with the UIMA Cas Visual Debugger application.

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

9.5 Get some Input Text

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

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

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

9.6 Run the Job

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

Note: The classpath for the application, defined in DuccRawTextSpec.job, requires that environmental parameters $UIMA_HOME and $OPENNLP_HOME are pointing at a valid UIMA SDK and the installed OpenNLP PEAR file, respectively.

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

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

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

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

9.7 Job Output

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

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

This output data will be used as input data for the following CAS input processing sample application.

9.8 Job Performance Details

DUCC captures a number of process performance metrics. Figure 9.1 shows details on the JD and single JP processes. The %CPU time shown, 728, is lower than the actual because the Job Process was idle for some time before it received the first Work Item and also idle between finishing the last Work Item and being shut down. DUCC shows the JVM spent a total of 58 seconds in GC (garbage collection), had no major page faults or page space, and used a max of 2.1GB of RSS.

Id: 8284 Workitems: 100 Done: 100 Error: 0 Dispatch: 0 Unassigned: 0 Limbo: 0

Processes | Work Items | Performance | Specification | Files

Processes List
click column heading to sort

Id	Log	Size	Host Name	PID	State Scheduler	Reason Scheduler or extraordinary status	State Agent	Reason Agent	Time Init	Time Run	Time GC	PgIn	Swap	%CPU	RSS
0	jd.out.log	0.06	allocj532.allocj.net	2307	Deallocated	Voluntary	Stopped		00	15:59	00	0	0.0	1.0	0.1
407	8284-UIMA-allocj537.allocj.net-2129.log	0.33	allocj537.allocj.net	2129	Deallocated		Running		19	15:05	58	0	0.0	728.0	2.1

Figure 9.1: OpenNLP Process Measurements

On the Performance tab, DUCC shows the breakdown of clock time spent in each primitive UIMA component running in the Job Process. See Figure 9.2. Processing time was dominated by the Parser component at 76.7%. The time spent compressing and writing out CASes was 0.5%, and the time reading the input text files well below 0.1%.

Id: 8284 Workitems: 100 Done: 100 Error: 0

Processes | Work Items | Performance | Specification | Files

Name	Total ddd:hh:mm:ss	% of Total	Avg hh:mm:ss/workitem	Min hh:mm:ss/workitem	Max hh:mm:ss/workitem
Total	01:48:23	100.0	01:05	03	01:34
Parser	01:23:08	76.7	49	02	01:05
Chunker	09:09	8.5	05	00	11
Percentage Name Finder	02:09	2.0	01	00	02
Person Name Finder	01:55	1.8	01	00	02
Location Name Finder	01:54	1.8	01	00	02
Money Name Finder	01:51	1.7	01	00	02
Organization Name Finder	01:50	1.7	01	00	02
Time Name Finder	01:48	1.7	01	00	02
Date Name Finder	01:46	1.6	01	00	01
POS Tagger	01:32	1.4	00	00	01
Tokenizer	38	0.6	00	00	00
DuccCasCC	34	0.5	00	00	00
DuccTextCM	01	0.0	00	00	00
Sentence Detector	01	0.0	00	00	00
DUCC Job Flow Controller	00	0.0	00	00	00

Figure 9.2: OpenNLP Process Breakdown

Chapter 10

Sample Application: CAS Input Processing

10.1 Application Function and Design

The main purpose of this application is to demonstrate the overhead of processing a collection of CASes grouped into zipfiles and stored as ZIP compressed XmiCas or with UIMA compressed binary form 6 format.

Note: This application depends on successful processing of the work in the previous chapter.

10.2 Configuration Parameters

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

InputSpec path to directory containing input files (named InputSpec in the hope that more options will be added).

OutputDirectory path to directory for output files.

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

SendToLast (optional) boolean to route WorkItem CAS to last pipeline component. Set to true in this application.

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

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

XmiCompressionLevel (optional) compression value if using ZIP compression. Default is 7.

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

10.3 Run the Job

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

Note: The classpath for the application, defined in DuccCasInputSpec.job, requires that environmental parameters $UIMA_HOME and $OPENNLP_HOME are pointing at a valid UIMA SDK and the installed OpenNLP PEAR file, respectively.

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

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

10.4 Job Performance Details

Figure 10.1 shows the component breakdown using binary CAS compression. Reading and deserializing took 38% vs the 60% spent serializing and writing. Using 8 pipeline threads in one process the 19245 CASes output from the last application were read and re-written in 9 seconds.

Id: 8290 Workitems: 100 Done: 100 Error: 0 Dispatch: 0 Unassigned: 0 Limbo: 0

Processes | Work Items | Performance | Specification | Files

Name	Total ddd:hh:mm:ss	% of Total	Avg hh:mm:ss/workitem	Min hh:mm:ss/workitem	Max hh:mm:ss/workitem
Total	01:09	100.0	00	00	02
DuccCasCC	42	60.3	00	00	01
DuccCasCM	26	38.3	00	00	00
DUCC Job Flow Controller	00	1.3	00	00	00
DuccSampleAE	00	0.1	00	00	00

Figure 10.1: CAS Input Processing Performacne

10.5 Limiting Job Resources

Although this 8-threaded Job Process was primarily CPU bound doing serialization work, it is possible to become I/O bound with enough threads banging on a shared filesystem. DuccCasInputSpec.job demonstrates how to limit the total number of processing threads to 32 using the combination of process_thread_count=8 and process_deployments_max=4.

I/O vs CPU bottlenecks can be detected using the detailed performance job data reported by DUCC and comparing results with various levels of scale out.

Part IV

Ducc Administrators Guide

Chapter 11

Installation, Configuration, and Verification

11.1 Overview

DUCC is a multi-user, multi-system distributed application. The instuctions below follow a staged installation/verification methodology, roughly as follows:

- Single system installation.
- Add new machines to DUCC control.
- Enable processes to run with the credentials of multiple submitting user. This step requires root authority on one or more machines.
- Enable CGroup containers. This step requires root authority on every DUCC machine.

DUCC is distributed as a compressed tar file. The instructions below assume installation from one of this distribution media. If building from source, the build creates this file in your svn trunk/target directory. The distribution file is in the form

```
uima-ducc-[version]-bin.tar.gz
```

where [version] is the DUCC version; for example, *uima-ducc-VERSION-bin.tar.gz* (where VERSION is the current DUCC version). This document will refer to the distribution file as the "<distribution.file>".

11.2 Software Prerequisites

Single system installation:

- Reasonably current Linux. DUCC has been tested on SLES 11.x and RHEL 6.x

  Note: On some systems the default *user limits* for max user processes (ulimit -u) and nfiles (ulimit -n) are defined too low for DUCC. The shell login profile for user *ducc* should set the soft limit for max user processes to be the same as the hard limit (ulimit -u 'ulimit -Hu'), and the nfiles limit raised above 1024 to at least twice the number of user processes running on the cluster.

- Python 2.x, where 'x' is 4 or greater. DUCC has not been tested on Python 3.x.
- Java 7. DUCC has been tested and run using IBM and Oracle JDK 1.7.
- Passwordless ssh for user running DUCC

Additional requirements for multiple system installation:

- All systems must have a shared filesystem (such as NFS or GPFS) and common user space. The $DUCC_HOME directory must be located on a shared filesystem.

Additional requirements for running multiple user processes with their own credentials.

- A userid *ducc*, and group *ducc*. User *ducc* must the the only member of group *ducc*.
- DUCC run with user *ducc* credentials.
- Root access is required to setuid-root the DUCC process launcher.

Additional requirements for CGroup containers:

- A userid *ducc*, and group *ducc*. User *ducc* must the the only member of group *ducc*.
- DUCC run with user *ducc* credentials.
- libcgroup1-0.37+ on SLES and libcgroup-0.37+ on RHEL, along with a custom /etc/cgconfig.conf

In order to build DUCC from source the following software is also required:

- A Subversion client, from `http://subversion.apache.org/packages.html`. The svn url is `https://svn.apache.org/repos/asf/uima/sandbox/uima-ducc/trunk`.
- Apache Maven, from `http://maven.apache.org/index.html`

The DUCC webserver server optionally supports direct "jconsole" attach to DUCC job processes. To install this, the following is required:

- Apache Ant, any reasonably current version.

To (optionally) build the documentation, the following is also required:

- Latex, including the *pdflatex* and *htlatex* packages. A good place to start if you need to install it is `https://www.tug.org/texlive/`.

More detailed one-time setup instructions for source-level builds via subversion can be found here: `http://uima.apache.org/one-time-setup.html#svn-setup`

11.3 Building from Source

To build from source, ensure you have Subversion and Maven installed. Extract the source from the SVN repository named above.

Then from your extract directory into the root directory (usually current-directory/trunk), and run the command

```
mvn install
```

or

```
mvn install -Pbuild-duccdocs
```

if you have LaTeX insalled and wish to do the optional build of documentation.

If this is your first Maven build it may take quite a while as Maven downloads all the open-source pre-requisites. (The pre-requisites are stored in the Maven repository, usually your $HOME/.m2).

When build is complete, a tarball is placed in your current-directory/trunk/target directory.

11.4 Documentation

After installation the DUCC documentation is found (in both PDF and HTML format) in the directory ducc_runtime/docs. As well, the DUCC webserver contains a link to the full documentation on each

major page. The API is documented only via JavaDoc, distributed in the webserver's root directory `$DUCC_HOME/webserver/root/doc/api.`

If building from source, Maven places the documentation in

- `trunk/uima-ducc-duccdocs/target/site` (main documentation), and
- `trunk/target/site/apidocs` (API Javadoc)

11.5 Single System Installation and Verification

Although any user ID can be used to run DUCC, it is recommended to create user "ducc" to later enable use of cgroups as well as running processes with the credentials of the submitting user.

If multiple nodes are going to be added later, it is recommended to install the ducc runtime tree on a shared filesystem so that it can be mounted on the additional nodes.

Verification submits a very simple UIMA pipeline for execution under DUCC. Once this is shown to be working, one may proceed installing additional features.

11.6 Minimal Hardware Requirements for Single System Installation

- One Intel-based or IBM Power-based system. (More systems may be added later.)
- 8GB of memory. 16GB or more is preferable for developing and testing applications beyond the non-trivial.
- 1GB disk space to hold the DUCC runtime, system logs, and job logs. More is usually needed for larger installations.

Please note: DUCC is intended for scaling out memory-intensive UIMA applications over computing clusters consisting of multiple nodes with large (16GB-256GB or more) memory. The minimal requirements are for initial test and evaluation purposes, but will not be sufficient to run actual workloads.

11.7 Single System Installation

1. Expand the distribution file with the appropriate umask:

   ```
   (umask 022 && tar -zxf <distribution.file>)
   ```

   This creates a directory with a name of the form "apache-uima-ducc-[version]".

   This directory contains the full DUCC runtime which you may use "in place" but it is highly recommended that you move it into a standard location on a shared filesystem; for example, under ducc's HOME directory:

   ```
   mv apache-uima-ducc-[version] /home/ducc/ducc_runtime
   ```

   We refer to this directory, regardless of its location, as $DUCC_HOME. For simplicity, some of the examples in this document assume it has been moved to /home/ducc/ducc_runtime.

2. Change directories into the admin sub-directory of $DUCC_HOME:

   ```
   cd $DUCC_HOME/admin
   ```

3. Run the post-installation script:

   ```
   ./ducc_post_install
   ```

If this script fails, correct any problems it identifies and run it again.

Note that *ducc_post_install* initializes various default parameters which may be changed later by the system administrator. Therefore it usually should be run only during this first installation step.

4. If you wish to install jconsole support from the webserver, make sure Apache Ant is installed, and run

   `./sign_jconsole_jar`

   This step may be run at any time if you wish to defer it.

That's it, DUCC is installed and ready to run. (If errors were displayed during ducc_post_install they must be corrected before continuing.)

The post-installation script performs these tasks:

1. Verifies that the correct level of Java and Python are installed and available.
2. Creates a default nodelist, $DUCC_HOME/resources/ducc.nodes, containing the name of the node you are installing on.
3. Defines the "ducc head" node to be to node you are installing from.
4. Sets up the default https keystore for the webserver.
5. Installs the DUCC documentation "ducc book" into the DUCC webserver root.
6. Builds and installs the C program, "ducc_ling", into the default location.
7. Ensures that the (supplied) ActiveMQ broker is runnable.

11.8 Initial System Verification

Here we verify the system configuration, start DUCC, run a test Job, and then shutdown DUCC.

To run the verification, issue these commands.

1. cd $DUCC_HOME/admin
2. ./check_ducc

   Examine the output of check_ducc. If any errors are shown, correct the errors and rerun check_ducc until there are no errors.
3. Finally, start ducc: ./start_ducc

Start_ducc will first perform a number of consistency checks. It then starts the ActiveMQ broker, the DUCC control processes, and a single DUCC agent on the local node.

You will see some startup messages similar to the following:

```
ENV: Java is configured as: /share/jdk1.7/bin/java
ENV: java full version "1.7.0_40-b43"
ENV: Threading enabled: True
MEM: memory is 15 gB
ENV: system is Linux
allnodes /home/ducc/ducc_runtime/resources/ducc.nodes
Class definition file is ducc.classes
OK: Class and node definitions validated.
OK: Class configuration checked
Starting broker on ducchead.biz.org
Waiting for broker ..... 0
Waiting for broker ..... 1
ActiveMQ broker is found on configured host and port: ducchead.biz.org:61616
```

```
Starting 1 agents
********** Starting agents from file /home/ducc/ducc_runtime/resources/ducc.nodes
Starting warm
Waiting for Completion
ducchead.biz.org Starting rm
     PID 14198
ducchead.biz.org Starting pm
     PID 14223
ducchead.biz.org Starting sm
     PID 14248
ducchead.biz.org Starting or
     PID 14275
ducchead.biz.org Starting ws
     PID 14300
ducchead.biz.org
    ducc_ling OK
    DUCC Agent started PID 14325
All threads returned
```

Now open a browser and go to the DUCC webserver's url, http://<hostname>:42133 where <hostname> is the name of the host where DUCC is started. Navigate to the Reservations page via the links in the upper-left corner. You should see the DUCC JobDriver reservation in state WaitingForResources. In a few minutes this should change to Assigned. Now jobs can be submitted.

To submit a job,

1. $DUCC_HOME/bin/ducc_submit –specification $DUCC_HOME/examples/simple/1.job

Open the browser in the DUCC jobs page. You should see the job progress through a series of transitions: Waiting For Driver, Waiting For Services, Waiting For Resources, Initializing, and finally, Running. You'll see the number of work items submitted (15) and the number of work items completed grow from 0 to 15. Finally, the job will move into Completing and then Completed..

Since this example does not specify a log directory DUCC will create a log directory in your HOME directory under

`$HOME/ducc/logs/job-id`

In this directory, you will find a log for the sample job's JobDriver (JD), JobProcess (JP), and a number of other files relating to the job.

This is a good time to explore the DUCC web pages. Notice that the job id is a link to a set of pages with details about the execution of the job.

Notice also, in the upper-right corner is a link to the full DUCC documentation, the "DuccBook".

Finally, stop DUCC:

1. cd $DUCC_HOME/admin
2. ./stop_ducc -a

11.9 Add additional nodes to the DUCC cluster

Additional nodes must meet all *prerequisites*.

$DUCC_HOME must be on a shared filesystem and mounted at the same location on all DUCC nodes.

If user's home directories are on local filesystems the location for user logfiles should be specified to be on a shared filesystem.

Addional nodes are normally added to a worker node group. Note that the DUCC head node does not have to be a worker node. In addition, the webserver node can be separate from the DUCC head node (see webserver configuration options in ducc.properties).

For worker nodes DUCC needs to know what node group each machine belongs to, and what nodes need an Agent process to be started on.

The configuration shipped with DUCC have all nodes in the same "default" node pool. Worker nodes are listed in the file

`$DUCC_HOME/resources/ducc.nodes.`

During initial installation, this file was initialized with the node DUCC is installed on. Additional nodes may be added to the file using a text editor to increase the size of the DUCC cluster.

11.10 Ducc_ling Configuration - Running with credentials of submitting user

DUCC launches user processes through ducc_ling, a small native C application. By default the resultant process runs with the credentials of the user ID of the DUCC application. It is possible for multiple users to submit work to DUCC in this configuration, but it requires that the user ID running DUCC has write access to all directories to which the user process outputs data. By configuring the ducc user ID and ducc_ling correctly, work submitted by all users will run with their own credentials.

Before proceeding with this step, please note:

- The sequence operations consisting of *chown* and *chmod* MUST be performed in the exact order given below. If the *chmod* operation is performed before the *chown* operation, Linux will regress the permissions granted by *chmod* and ducc_ling will be incorrectly installed.

ducc_ling is designed to be a setuid-root program whose function is to run user processes with the identity of the submitting user. This must be installed correctly; incorrect installation can prevent jobs from running as their submitters, and in the worse case, can introduce security problems into the system.

ducc_ling can either be installed on a local disk on every system in the DUCC cluster, or on a shared-filesystem that does not suppress setuid-root permissions on client nodes. The path to ducc_ling must be the same on each DUCC node. The default path configuration is $DUCC_HOME/admin/${os.arch}/ in order to handle clusters with mixed OS platforms. ${os.arch} is the architecture specific value of the Java system property with that name; examples are amd64 and ppc64.

The steps are: build ducc_ling for each node architecture to be added to the cluster, copy ducc_ling to the desired location, and then configure ducc_ling to give user ducc the ability to spawn a process as a different user.

In the example below ducc_ling is left under $DUCC_HOME, where it is built.

As user *ducc*, build ducc_ling for necessary architectures (this is done automatically for the DUCC head machine by the ducc_post_install script). For each unique OS platform:

1. cd $DUCC_HOME/admin
2. ./build_duccling

Then, as user *root* on the shared filesystem, *cd $DUCC_HOME/admin*, and for each unique OS architecture:

1. chown ducc.ducc ${os.arch}
   (set directory ownership to be user ducc, group ducc)
2. chmod 700 ${os.arch}
   (only user ducc can read contents of directory)
3. chown root.ducc ${os.arch}/ducc_ling
   (make root owner of ducc_ling, and let users in group ducc access it)

4. chmod 4750 ${os.arch}/ducc_ling
   (ducc_ling runs as user root when started by users in group ducc)

If these steps are correctly performed, ONLY user *ducc* may use the ducc_ling program in a privileged way. ducc_ling contains checks to prevent even user *root* from using it for privileged operations.

If a different location is chosen for ducc_ling the new path needs to be specified for ducc.agent.launcher.ducc_spawn_path in $DUCC_HOME/resources/site.ducc.properties. See more info at see Properties merging.

11.11 CGroups Installation and Configuration

Note: A key feature of DUCC is to run user processes in CGroups in order to guarantee each process always has the amount of RAM requested. RAM allocated to the managed process (and any child processes) that exceed requested DUCC memory size will be forced into swap space. Without CGroups a process that exceeds its requested memory size by N% is killed (default N=5 in ducc.properties), and memory use by child processes is ignored.

DUCC's CGroup configuration also allocates CPU resources to managed processes based on relative memory size. A process with 50% of a machine's RAM will be guaranteed at least 50% of the machine's CPU resources as well.

The steps in this task must be done as user root and user ducc.

To install and configure CGroups for DUCC:

1. Install the appropriate libcgroup package at level 0.37 or above.
2. Configure /etc/cgconfig.conf as follows:

```
# Mount cgroups
mount {
   memory = /cgroup;
   cpu = /cgroup;
}
# Define cgroup ducc and setup permissions
group ducc {
 perm {
     task {
        uid = ducc;
     }
     admin {
        uid = ducc;
     }
 }
 memory {}
 cpu{}
}
```

3. Start the cgconfig service:

```
service cgconfig start
```

4. Verify cgconfig service is running by the existence of directory:

```
/cgroups/ducc
```

5. Configure the cgconfig service to start on reboot:

```
chkconfig cgconfig on
```

Note: if CGroups is not installed on a machine the DUCC Agent will detect this and not attempt to use the feature. CGroups can also be disabled for all machines (see *ducc.agent.launcher.cgroups.enable*) or it can be disabled for individual machines (see *ducc.agent.exclusion.file*).

11.12 Full DUCC Verification

This is identical to initial verification, with the one difference that the job "1.job" should be submitted as any user other than ducc. Watch the webserver and check that the job executes under the correct identity. Once this completes, DUCC is installed and verified.

11.13 Enable DUCC webserver login

This step is optional. As shipped, the webserver is disabled for logins. This can be seen by hovering over the Login text located in the upper right of most webserver pages:

```
System is configured to disallow logins
```

To enable logins, a Java-based authenticator must be plugged-in and the login feature must be enabled in the ducc.properties file by the DUCC administrator. Also, ducc_ling should be properly deployed (see Ducc_ling Installation section above).

A beta version of a Linux-based authentication plug-in is shipped with DUCC. It can be found in the source tree:

```
org.apache.uima.ducc.ws.authentication.LinuxAuthenticationManager
```

The Linux-based authentication plug-in will attempt to validate webserver login requests by appealing to the host OS. The user who wishes to login provides a userid and password to the webserver via https, which in-turn are handed-off to the OS for a success/failure reply.

To have the webserver employ the beta Linux-based authentication plug-in, the DUCC administrator should perform the following as user ducc:

```
1. edit ducc.properties
2. locate: ducc.ws.login.enabled = false
3. modify: ducc.ws.login.enabled = true
4. save
```

Note: The beta Linux-based authentication plug-in has limited testing. In particular, it was tested using:

```
Red Hat Enterprise Linux Workstation release 6.4 (Santiago)
```

Alternatively, you can provide your own authentication plug-in. To do so:

```
1. author a Java class that implements
   org.apache.uima.ducc.common.authentication.IAuthenticationManager
2. create a jar file comprising your authentication class
3. put the jar file in a location accessible by the DUCC webserver, such as
    $DUCC_HOME/lib/authentication
4. put any authentication dependency jar files there as well
5. edit ducc.properties
6. add the following:
   ducc.local.jars = authentication/*
   ducc.authentication.implementer=<your.authenticator.class.Name>
7. locate: ducc.ws.login.enabled = false
8. modify: ducc.ws.login.enabled = true
9. save
```

Chapter 12

Administration

12.1 WebServer Authentication

By default, DUCC is configured such that there is effectively no authentication enforcement by the WebServer. No password entry is permitted on the Login panel and any userid specified is accepted whether it exists or not.

To enable your own authentication measures, you should perform the following steps:

1. Author an authentication manager Java class implementing interface

   `org.apache.uima.ducc.common.authentication.IAuthenticationManager`

2. Create an authentication jar file comprising the authentication manager Java class
3. Install your authentication jar file and any dependency jar files into your DUCC's lib folder
4. Update your ducc.properties file with authentication class name and jar file name(s) information
5. Create a ducc.administrators file

Note: When a user clicks on the WebServer Login link, the login dialog is shown. On that dialog panel is shown the authenticator: *version*, which is supplied by your authentication manager implementation's *getVersion()* method. Also shown are boxes for userid and password entry. If your authentication manager implemenation's *isPasswordChecked()* method returns true then the password box will accept input, otherwise it will be disabled.

12.1.1 Example Implementation

Shown below is an example implementation which can be used as a template for coding protection by means of interfacing with your site's security measures.

In this example, the SiteSecurity Java class is presumed to be existing and available code at your installation.

```
package org.apache.uima.ducc.example.authentication.module;

import org.apache.uima.ducc.common.authentication.AuthenticationResult;
import org.apache.uima.ducc.common.authentication.IAuthenticationManager;
import org.apache.uima.ducc.common.authentication.IAuthenticationResult;
import org.apache.uima.ducc.example.authentication.site.SiteSecurity;

public class AuthenticationManager implements IAuthenticationManager {

    private final String version = "example 1.0";
```

```
    @Override
    public String getVersion() {
        return version;
    }

    @Override
    public boolean isPasswordChecked() {
        return true;
    }

    @Override
    public IAuthenticationResult isAuthenticate(String userid, String domain,
            String password) {
        IAuthenticationResult authenticationResult = new AuthenticationResult();
        authenticationResult.setFailure();
        try {
            if(SiteSecurity.isAuthenticUser(userid, domain, password)) {
                authenticationResult.setSuccess();
            }
        }
        catch(Exception e) {
            //TODO
        }
        return authenticationResult;
    }

    @Override
    public IAuthenticationResult isGroupMember(String userid, String domain,
            Role role) {
        IAuthenticationResult authenticationResult = new AuthenticationResult();
        authenticationResult.setFailure();
        try {
            if(SiteSecurity.isAuthenticRole(userid, domain, role.toString())) {
                authenticationResult.setSuccess();
            }
        }
        catch(Exception e) {
            //TODO
        }
        return authenticationResult;
    }

}
```

12.1.2 IAuthenticationManager

Shown below is the interface which must be implemented by your authentication manager.

```
package org.apache.uima.ducc.common.authentication;

public interface IAuthenticationManager {

    /**
```

```
     * This method is expected to return AuthenticationManager implementation version
     * information.  It is nominally displayed by the DUCC webserver on the Login/Logout
     * pages.
     *
     * Example return value: Acme Authenticator 1.0
     *
     * @return The version of the AuthenticationManager implementation.
     */
    public String getVersion();

    /**
     * This method is expected to return password checking information.
     * It is nominally employed by the DUCC webserver to enable/disable
     *  password input area on the Login/Logout pages.
     *
     * @return True if the AuthenticationManager implementation checks passwords;
     * false otherwise.
     */
    public boolean isPasswordChecked();

    /**
     * This method is expected to perform authentication.
     * It is nominally employed by the DUCC webserver for submitted Login pages.
     *
     * @param userid
     * @param domain
     * @param password
     * @return True if authentic userid+domain+password; false otherwise.
     */
    public IAuthenticationResult isAuthenticate(String userid, String domain, String password);

    /**
     * This method is expected to perform role validation.
     * It is nominally employed by the DUCC webserver for submitted Login pages.
     *
     * @param userid
     * @param domain
     * @param role
     * @return True if authentic userid+domain+role; false otherwise.
     */
    public IAuthenticationResult isGroupMember(String userid, String domain, Role role);

    /**
     * The supported Roles
     */
    public enum Role {
        User,
        Admin
    }
}
```

12.1.3 IAuthenticationResult

Shown below is the interface which must be returned by the required authentication methods in your authentication manager.

```
package org.apache.uima.ducc.common.authentication;

public interface IAuthenticationResult {
    public void setSuccess();
    public void setFailure();
    public boolean isSuccess();
    public boolean isFailure();
    public void setCode(int code);
    public int getCode();
    public void setReason(String reason);
    public String getReason();
    public void setException(Exception exception);
    public Exception getException();
}
```

12.1.4 Example ANT script to build jar

Shown below is an example ANT script to build a ducc-authenticator.jar file. The resulting jar file should be placed user DUCC's lib directory along with any dependency jars, and defined in ducc.properties file.

```
<project name="uima-ducc-examples" default="build" basedir=".">

    <property name="TGT-LIB"                value="${basedir}/lib" />
    <property name="TGT-DUCC-AUTH-JAR"      value="${TGT-LIB}/ducc-authenticator.jar" />

    <target name="build" depends="clean, jar" />

    <target name="clean">
        <delete file="${TGT-DUCC-AUTH-JAR}" />
    </target>

    <target name="jar">
        <mkdir dir="${TGT-LIB}" />
        <jar destfile="${TGT-DUCC-AUTH-JAR}" basedir="${basedir}/target/classes/org/apache/uima/ducc/exampl
    </target>

</project>
```

12.1.5 Example ducc.properties entries

Shown here is a snippet of the ducc.properties file defining the class to be used for authentication and the administrator created folder *site-security*, which should contain the ducc-authenticator.jar you built plus any jar files upon which it depends.

Note: the *site-security* directory must be located within DUCC's lib directory.

```
# The class that performs authentication (for the WebServer)
ducc.authentication.implementer = org.apache.uima.ducc.example.authentication.module.AuthenticationManager
```

```
# Site specific jars: include all jars in directory site-security
ducc.local.jars = site-security/*
```

12.1.6 Example ducc.administrators

Example contents of ducc.administrators file located within DUCC's resources directory. Only userids listed here can assume the Administrator role when performing operations via the WebServer.

```
jdoe
fred
hal9000
```

12.2 Properties

Public properties are in a primary configuration file is called ducc.properties and always resides in the directory ducc_runtime/resources.

Private properties are in a secondary configuration file call ducc.private.properties and always resides in the directory ducc_runtime/resources/private.

12.3 Properties merging

With DUCC 2.0.0 the shipped DUCC properties file is designed to be read-only. Installations create a local properties file which is automatically merged with the default properties file as part of system startup.

The shipped DUCC properties file is called *default.ducc.properties*. This file should never be edited or modified.

The local site override properties file is called *site.ducc.properties*. This is a normal Java properties file containing override and additional properties. An initial *site.ducc.properties* is created on installation of DUCC 2.0.0 by *ducc_post_install*.

On startup (*start_ducc*), verification (*check_ducc*), and RM reconfiguration (*rm_reconfigure*), the two properties files are merged, with *site.ducc.properties* taking preference, to create the operational file, *ducc.properties*, which is used by all DUCC compononts. This file should not be edited as it will be over-written whenever *start_ducc* or *check_ducc* is run.

12.4 ducc.properties

Some of the properties in ducc.properties are intended as the "glue" that brings the various DUCC components together and lets then run as a coherent whole. These types of properties should be modified only by developers of DUCC itself. In the description below these properties are classified as "Private".

Some of the properties are tuning parameters: timeouts, heartbeat intervals, and so on. These may be modified by DUCC administrators, but only after experience is gained with DUCC, and only to solve specific performance problems. The default tuning parameters have been chosen by the DUCC system developers to provide "best" operation under most reasonable situations. In the description below these properties are classified as "Tuning".

Some of the properties describe the local cluster configuration: the location of the ActiveMQ broker, the location of the Java JRE, port numbers, etc. These should be modified by the DUCC administrators to configure DUCC to each individual installation. In the description below these properties are classified as "Local".

See also

12.4.1 General DUCC Properties

ducc.authentication.implementer

This specifies the class used for WebServer session authentication. If unconfigured, the Web Server enforces no authentication.

Default org.apache.uima.ducc.common.authentication.LinuxAuthenticationManager

Type Local

ducc.authentication.users.include

Specify users allowed to log in to the web server. This is used only if *ducc.authentication.implementor* is the LinuxAuthenticationManager.

Default All users may log in.

Type Local

ducc.authentication.users.exclude

Specify users not allowed to log in to the webserver. This is used only if *ducc.authentication.implementor* is the LinuxAuthenticationManager.

Default No users are excluded.

Type Local

ducc.authentication.groups.include

Specify groups allowed to log in. Groups are defined by Unix authentication. Only users in the groups specified here may log in to the web server. This is used only if *ducc.authentication.implementor* is the LinuxAuthenticationManager.

Default Users in all groups may log in.

Type Local

ducc.authentication.groups.exclude

Specify groups not allowed to log in. Groups are defined by Unix authentication. Users in the groups specified here may not log in to the web server. This is used only if *ducc.authentication.implementor* is the LinuxAuthenticationManager.

Default No users are excluded due to group membership.

Type Local

ducc.admin.endpoint

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

Default ducc.admin.channel

Type Private

ducc.admin.endpoint.type

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

Default topic

Type Private

ducc.broker.automanage

If set to “true”, DUCC will start and stop the ActiveMQ broker as part of its normal start/stop scripting.

Default true

Type Tuning

ducc.broker.configuration
This is the ActiveMQ configuration file to use, for auto-managed brokers only. The path must be specified relative to the ActiveMQ installation directory.

Default conf/activemq-ducc.xml

Type Tuning

ducc.broker.credentials
This is the ActiveMQ credentials file used to authenticate DUCC daemons with the broker, for auto-managed brokers only.

Default $ducc.private.resources/ducc-broker-credentials.properties

Type Tuning

ducc.broker.home
For DUCC auto-managed brokers only, this names the location where ActiveMQ is installed installed.

Note that the DUCC installation includes a default ActiveMQ.

Default $DUCC_HOME/activemq

Type Tuning

ducc.broker.memory.options
For DUCC auto-managed brokers only, this names the ActiveMQ configuration file. The configuration file is assumed to reside in the directory specified by *ducc.broker.home*, so the path must be relative to that location.

Default conf/activemq-ducc.xml

Type Tuning

ducc.broker.url.decoration
The property is used by the DUCC Job Driver processes to modify the ActiveMQ broker URL when connecting to the Job Processes.

The supplied default is used to disable broker connection timeouts. From the ActiveMQ documentation: "The maximum inactivity duration (before which the socket is considered dead) in milliseconds. On some platforms it can take a long time for a socket to appear to die, so we allow the broker to kill connections if they are inactive for a period of time. Use by some transports to enable a keep alive heart beat feature. Set to a value less-than-or-equal0 to disable inactivity monitoring. Declare the wire protocol used to communicate with ActiveMQ."

This decoration is used to keep the broker connection alive while a JVM is in a long garbage collection. The applications that DUCC is designed to support can spend significant time in garbage collection, which can cause spurious timeouts. By default the DUCC configuration disables the timeout by setting it to 0.

Default wireFormat.maxInactivityDuration=0

Type Local

ducc.broker.hostname
This declares the node where the ActiveMQ broker resides. It MUST be updated to the actual node where the broker is running as part of DUCC installation. The default value will not work.

Default ${ducc.head}. The default is defined in the ducc property, *ducc.head*. If you want to run the ActiveMQ broker on the "ducc head", this parameter need not be changed.

Type Local

ducc.broker.jmx.port
This is the port used to make JMX connections to the broker. This should only be changed by administrators familiar with ActiveMQ configuration.

Default 1100

Type Local

ducc.broker.memory.options
For DUCC auto-managed brokers only, this sets the `-Xmx` heap size for the broker.

Default -Xmx2G

Type Tuning

ducc.broker.name
This is the internal name of the broker, used to locate Broker's MBean in JMX Registry. It is NOT related to any node name. When using the ActiveMQ distribution supplied with DUCC it should always be set to "localhost". The default should be changed only by administrators familiar with ActiveMQ configuration.

Default localhost

Type Local

ducc.broker.port
This declares the port on which the ActiveMQ broker is listening for messages. It MAY be updated as part of DUCC installation. ActiveMQ ships with port 61616 as the default port, and DUCC uses that default.

Default 61617

Type Local

ducc.broker.protocol
Declare the wire protocol used to communicate with ActiveMQ.

Default tcp

Type Private

ducc.broker.server.url.decoration
For DUCC auto-managed brokers only, this configures ActiveMQ Server url decoration.

Default transport.soWriteTimeout=45000

Type Tuning

ducc.cli.httpclient.sotimeout
This is the timeout used by the CLI to communicate with DUCC, in milliseconds. If no response is heard within this time, the request times out and is aborted. When set to 0 (the default), the request never times out.

Default 0

Type Tuning

ducc.cluster.name
This is a string used in the Web Server banner to identify the local cluster. It is used for informational purposes only and may be set to anything desired.

Default Apache UIMA-DUCC

Type Local

ducc.head
This property declares the node where the DUCC adminstrative processes run (Orchestrator, Resource Manager, Process Manager, Service Manager). This property is required and MUST be configured in new installation. The installation script ducc_post_install initializes this property to the node the script is executed on.

Default There is no default, this must be configured during system installation.

Type Local

ducc.jms.provider
Declare the type of middleware providing the JMS service used by DUCC.

Default activemq

Type Private

ducc.jmx.port
Every process started by DUCC has JMX enabled by default. When more than one process runs on the same machine this can cause port conflicts. The property "ducc.jmx.port" is used as the base port for JMX. If the port is busy, it is incremented internally until a free port is found.

The web server's "System −> Daemons" tab is used to find the JMX URL that gets assigned to each of the DUCC management processes. The web server's Job details page for each job is used to find the JMX URL that is assigned to each JP.

Default 2099

Type Private

ducc.jvm
This specifies the full path to the JVM to be used by the DUCC processes. This MUST be configured. The installation script ducc_post_install initializes this property to full path to "java" in the installer's environment. (If the "java" command cannot be found, ducc_post_install exits with error.)

Default None. Must be configured during installation.

Type Local

ducc.node.min.swap.threshold
Specify a minimum amount of free swap space available on a node. If an agent detects free swap space dipping below the value defined below, it will find the fattest (in terms of memory) process in its inventory and kill it. The value of the parameter below is expressed in bytes.

If set to 0, the threshold is disabled.

Default 0

Type Tuning

ducc.agent.jvm.args
This specifies the list of arguments passed to the JVM when spawning the Agent.

Default -Xmx100M

Type Tuning

ducc.driver.jvm.args
If enabled, the arguments here are automatically added to the JVM arguments specified for the Job Driver process.

Note: if the user-supplied JVM arguments contain a -Xmx entry then any -Xmx value specified here will be ignored.

Default (unconfigured)

Type Local

ducc.driver.jetty.max.threads
Max number of threads in Jetty thread pool servicing incoming HTTP requests.

Default 100

Type Tuning

ducc.driver.jetty.thread.idletime
Max idle time for jetty threads (in milliseconds). When a thread exceeds its idle time it will be terminated.

Default 60000

Type Tuning

ducc.orchestrator.jvm.args
This specifies the list of arguments passed to the JVM when spawning the Orchestrator.

Default -Xmx1G

Type Tuning

ducc.pm.jvm.args
This specifies the list of arguments passed to the JVM when spawning the Process Manager.

Default -Xmx1G

Type Tuning

ducc.process.jvm.args
If enabled, the arguments here are added by DUCC to the JVM arguments in the user's job processes.

Default (unconfigured)

Type Private

ducc.rm.jvm.args
This specifies the list of arguments passed to the JVM when spawning the Resource Manager.

Default -Xmx1G

Type Tuning

ducc.sm.jvm.args
This specifies the list of arguments passed to the JVM when spawning the Service Manager.

Default -Xmx1G

Type Tuning

ducc.ws.jvm.args
specifies the list of arguments passed to the JVM when spawning the Webserver.

Default -Xmx8G

Type Tuning

ducc.locale.language
Establish the language for national language support of messages. Currently only ”en” is supported.

Default en

Type Private

ducc.locale.country
Establish the country for National Language Support of messages. Currently only ”us” is supported.

Default us

Type Private

ducc.runmode
When set to ”Test” this property bypasses userid and authentication checks. It is intended for use ONLY by DUCC developers. It allows developers of DUCC to simulate a multiuser environment without the need for root privileges.

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

Default Unconfigured. When unconfigured, test mode is DISABLED.

Type Local

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

Default on

Type Tuning

ducc.threads.limit
This enforces a maximum number of pipeline threads per job, over all its processes. No job will have more active work-items than this dispatched. This limit is disabled by default.

The value represents the size of the underlying CAS pool in the Job Driver and therefore is related to the size of the Job Driver heap and the real memory consumed by JD. If the JD is consuming too much memory, try setting or reducing this value.

Default (unconfigured)

Type Local

ducc.environment.propagated
This specifies the environmental variables whose values will be merged with the user-specified environment option on job, process and service submissions.

Default USER HOME LANG DUCC_SERVICE_INSTANCE

Type Local

12.4.2 Web Server Properties

ducc.ws.configuration.class
The name of the pluggable java class used to implement the Web Server.

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

Type Private

ducc.ws.node
This is the name of the node the web server is started on. If not specified, the web server is started on `${ducchead}`.

Default Value (unconfigured)

Type Local

ducc.ws.ipaddress
In multi-homed systems it may be necessary to specify to which of the multiple addresses the Web Server listens for requests. This property is an IP address that specifies to which address the Web Server listens.

Default Value (unconfigured)

Type Local

ducc.ws.port
This is the port on which the DUCC Web Server listens for requests.

Default Value 42133

Type Local

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

Default Value 42155

Type Local

ducc.ws.session.minutes
Once authenticated, this property determines the lifetime of the authenticated session to the Web Server.

Default Value 60

Type Tuning

ducc.ws.max.history.entries
DUCC maintains a history of all jobs. The state of jobs, both old and current are shown in the Webserver's Jobs Page. To avoid overloading this page and the Web Server, the maximum number of entries that can be shown is regulated by this parameter.

Default Value 4096

Type Tuning

ducc.ws.login.enabled
If true, users are allowed to login to Webserver. If false, users are not allowed to login to Webserver. Shipped value set to false. However, default value if property not specified is true.

Default Value true

Type Tuning

ducc.ws.precalculate.machines
This is a choice between updating the sorted internal representation of the Machines page as each Agent publication arrives (true, somewhat CPU intensive in a large cluster but fast browser response time) and updating only upon viewer demand (false, CPU intensive for each browser request with slower response time).

Default Value true

Type Tuning

ducc.ws.automatic.cancel.minutes Optionally configure the webserver job automatic cancel timeout. To disable this feature specify 0. This is employed when a user specifies −−*wait_for_completion* flag on job submission, in which case the job monitor program must visit

```
http://<host>:<port>/ducc-servlet/proxy-job-status?id=<job-id>
```

within this expiry time. Otherwise the job will be automatically canceled.

This provides a safeguard against runaway jobs or managed reservations, if the submitter gets disconnected from DUCC in some way.

If the feature is disabled by specifing "0", no work is canceled even if the monitor itself disappears.

Default Value 10

Type Tuning

ducc.ws.jsp.compilation.directory
This specifies the temporary used by the Web Server's JSP engine to compile its JSPs.

Default Value /tmp/ducc/jsp

Type Tuning

ducc.ws.requestLog.RetainDays
Optionally configure the webserver request log, default, if not configured, is 0 (meaning no request logging). Logs are written to DUCC_HOME/logs/webserver.

Default Value 30

Type Tuning

ducc.ws.visualization.strip.domain
If set, the visualization will strip domain names from nodes to present a cleaner visualization.

Default Value true

Type Tuning

12.4.3 Job Driver Properties

ducc.jd.configuration.class
The name of the pluggable java class used to implement the Job Driver.

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

Type Private

ducc.jd.state.update.endpoint
This is the JMS endpoint name by the Job Driver to send state to the Orchestrator.

Default Value ducc.jd.state

Type Private

ducc.jd.startup.initialization.error.limit
For a newly started Job, the number of JP UIMA initialization failures allowed until at least one JP succeeds - otherwise, the Job self-destructs.

Default Value 1

Type Tuning

ducc.jd.state.update.endpoint.type
This is the JMS message type used to send state to the Orchestrator.

Default Value topic

Type Private

ducc.jd.state.publish.rate
The interval in milliseconds between JD state publications to the Orchestrator. A higher rate (smaller number) may slightly increase system response but will increase network load. A lower rate will somewhat decrease system response and lower network load.

Default Value 15000

Type Tuning

ducc.jd.queue.prefix
This is a human-readable string used to form queue names for the JMS queues used to pass CASs from the Job Driver to the Job Processes. The completed queue named comprises the prefix concatenated with the DUCC assigned Job number.

Default Value ducc.jd.queue.

Type Private

ducc.jd.queue.timeout.minutes
After dispatching a work item to UIMA-AS client for processing, the number of minutes that the Job Driver will wait for two callbacks (queued and assigned) before considering the work item lost. The elapsed time for the callbacks is normally sub-second. Intermittent network problems may cause unusual spikes.

Default Value 5

Type Tuning

ducc.jd.share.quantum
When CGroups are enabled, this is the RSS, in MB, that is reserved for each JD process, and enforced by the CGroup support. Larger JDs are permitted, but the CGroup support will force the excess RSS onto swap. This potentially slows the performance of that JD, but preserves the resources for other, better-behaved, JDs.

Default Value 400

Type Tuning

12.4.4 Service Manager Properties

ducc.sm.configuration.class
This is the name of the pluggable java class used to implement the Service Manager.

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

Type Private

ducc.sm.default.monitor.class
This is the name of the default UIMA-AS ping/monitor class. The default class issues *get-meta* to a service and uses JMX to fetch queue statistics for presentation in the web server.

This name is either

1. The fully qualified name of the class to use as the default UIMA-AS pinger. It may be necessary to include the class or jar file in the classpath used to start the SM. (The reccomended way to do this is add an entry to the *ducc.local.jars* property in *ducc.properties.*
2. The name of a pinger registration file. This is the reccomended way to provide installation-customized pingers. See the Service Management chapter for details of setting up this file. In short, it resides in *ducc.properties* and contains the full set of ping-related properties needed to run a pinger.

Default Value org.apache.uima.ducc.cli.UimaAsPing

Type Tuning

ducc.sm.state.update.endpoint
This is the JMS endpoint name used for state messages sent by the Service Manager.

Default Value ducc.sm.state

Type Private

ducc.sm.state.update.endpoint.type
This is the JMS message type used for state messages sent by the Service Manager.

Default Value topic

Type Private

ducc.sm.meta.ping.rate
This is the time, in milliseconds, between pings by the Service Manager to each known, running service.

Default Value 60000

Typ Tuning

ducc.sm.meta.ping.stability
This is the number of consecutive pings that may be missed before a service is considered unavailable.

Default Value 10

Type Tuning

ducc.sm.meta.ping.timeout
This is the time in milliseconds the SM waits for a response to a ping. If the service does not respond within this time the ping is accounted for as a "missed" ping.

Default Value 15000

Type Tuning

ducc.sm.http.port
This is the HTTP port used by the Service Manager to field requests from the CLI / API.

Default Value 19989

Type Local

ducc.sm.http.node
This is the node where the Service Manager runs. It MUST be configured as part of DUCC setup. The *ducc_post_install* procedures initialize this to *${ducc.head}*.

Default Value ${ducc.head}

Type Local

ducc.sm.default.linger
This is the length of time, in milliseconds, that the SM allows a service to remain alive after all jobs that reference it have exited. If no new job referencing it enters the system before this time has expired, the SM stops the service.

Default Value 300000

Type Tuning

ducc.sm.init.failure.limit
This is the maximum number of consecutive failures of service instance initialization permitted before DUCC stops creating new instances. When this cap is hit the SM will disable autostart for the service. It may be overridden by the service registration's *instance_failures_limit* parameter.

NOTE: This was *ducc.sm.instance.failure.max* which is now deprecated.

Default Value 2

Type Tuning

ducc.sm.instance.failure.limit
This is the maximum number of instance failures allowed within some period of time before the Service Manger disables *autostart* and ceases to restart instances automatically. The time window for failures is defined with the property *ducc.sm.instance.failure.window*.

This may be overridden by individual service pingers using the registration property *instance_failures_limit*. See the service registration options for details.

Default Value 5

Type Tuning

ducc.sm.instance.failure.window
This specifies a window of time in minutes over which some number of service instance failures are tolerated. If the maximum number of tolerated failures is exceeded within this time window the Service Manager ceases to restart instances automatically. The maximum tolerated failures is defined in *ducc.sm.instance.failure.limit*.

This may be overridden by individual service pingers using the registration property *instance_failures_window*. See the service registration options for details.

Default Value 30 (minutes)

Type Tuning

12.4.5 Orchestrator Properties

ducc.orchestrator. configuration.class
This is the name of the pluggable java class used to implement the DUCC Orchestrator.

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

Type Private

ducc.orchestrator.start.type
This indicates the level of recovery to be taken on restarting a system. There are two levels of startup:

cold All reservations are canceled, all currently running jobs (if any) are terminated. All services are terminated. The system starts with no jobs, reservations, or services active.

warm All unmanaged reservations are restored. All currently running jobs (if any) are terminated. All services are started or restarted as indicated by their state when the system went down. The system starts with no jobs active, but unmanaged reservations and services are preserved.

Default Value warm

Type Tuning

ducc.orchestrator.state.endpoint
This is the name of the JMS endpoint through which the Orchestrator broadcasts its full state messages. These messages include full job information and can be large. This state is used by the Process Manager and the Webserver.

Default Value ducc.orchestrator.request?requestTimeout=180000

Type Private

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

Default Value topic

Type Private

ducc.orchestrator.state.publish.rate

The interval in milliseconds between Orchestrator publications of its non-abbreviated state.

Default Value 10000

Type Private

ducc.orchestrator.maintenance.rate
This is the interval in milliseconds between Orchestrator maintenance cycles, which check and update history and state.

Default Value 60000

Type Tuning

ducc.orchestrator.http.port
This is the HTTP port used by the Orchestrator to field requests from the CLI / API.

Default Value 19988

Type Local

ducc.orchestrator.http.node
This is the node where the Orchestrator runs. It MUST be configured as part of DUCC setup. The *ducc_post_install* procedures initialize this to *${ducc.head}*.

Default Value ${ducc.head}

Type Local

ducc.orchestrator.unmanaged.reservations.accepted
This flag controls whether the Orchestrator will accept requests for unmanaged reservations (true) or deny request for unmanaged reservations (false).

Default Value true

Type Local

12.4.6 Resource Manager Properties

ducc.rm.configuration.class
This is the name of the pluggable java class used to implement the DUCC Resource Manager.

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

Type Private

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

Default Value ducc.rm.state

Type Private

ducc.rm.state.update.endpoint.type
This is the JMS endpoint type used for state messages sent by the Resource Manager.

Default Value topic

Type Private

ducc.rm.state.publish.ratio
This specifies the frequency of RM schedules, relative to the number of Orchestrator publications. If the value is set to 1, RM runs and publishes a schedule immediately on receipt of OR state. If set to some number N, RM runs a schedule after receipt of every N Orchestrator publications.

Default Value 1

Type Tuning

ducc.rm.share.quantum
The share quantum is the smallest amount of RAM that is schedulable for jobs, in GB. Jobs are scheduled based entirely on their memory requirements. Memory is allocated in multiples of the share quantum.

See the Resource Management section for more information on the share quantum.

Default Value 1

Type Tuning

ducc.rm.global.allotment
This specifies the maximum non-preemptable shares any user may be awarded, in GB. If not configured, there is no maximum enforced. This can be overridden on a per-user basis in the user registry. See the Resource Management section for more information on the share quantum.

Default Value (not set, no limit imposed)

Type Tuning

ducc.rm.scheduler
The component that implements the scheduling algorithm is pluggable. This specifies the name of that class.

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

Type Private

ducc.rm.user.registry

This names the file with the user registr, within the DUCC_HOME/resources directoryy. As of this version of DUCC, the registry is used only to override the global allotments. The registry entries may also be placed in the *ducc.classes* file if desired.

Default Value ducc.users

Type Private

ducc.rm.class.definitions

This specifies the name of the file that contains the site's class definitions. This file is described in detail in the *ducc.classes* section.

Default Value ducc.classes

Type Tuning

ducc.rm.node.stability

The RM receives regular "heartbeats" from the DUCC agents in order to know what nodes are available for scheduling. The node.stability property configures the number of consecutive heartbeats that may be missed before the Resource Manager considers the node to be inoperative.

If a node becomes inoperative, the Resource Manager deallocates all processes on that node and attempts to reallocate them on other nodes. The node is marked offline and is unusable until its heartbeats start up again.

The default configuration declares the agent heartbeats to occur at 1 minute intervals. Therefore heartbeats must be missed for five minutes before the Resource Manager takes corrective action.

Default Value 5

Type Tuning

ducc.rm.init.stability

During DUCC initialization the Resource Manager must wait some period of time for all the nodes in the cluster to check-in via their "heartbeats". If the RM were to start scheduling too soon there would be a period of significant "churn" as the perceived cluster configurations changes rapidly. As well, it would be impossible to recover work in a warm or hot start if the affected nodes had not yet checked in.

The init.stability property indicates how many heartbeat intervals the RM must wait before it starts scheduling after initialization.

Default Value 2

Type Tuning

ducc.rm.eviction.policy

The alternative value is SHRINK_BY_MACHINE.

The eviction.policy is a heuristic to choose which processes of a job to preempt because of competition from other jobs.

The SHRINK_BY_INVESTMENT policy attempts to preempt processes such that the least amount of work is lost. It chooses candidates for eviction in order of:

1. Processes still initializing, with the smallest time spent in the initializing step.
2. Processes whose currently active work items have been executing for the shortest time.

The SHRINK_BY_MACHINE policy attempts to preempt processes so as to minimize fragmentation on machines with large memories that can contain multiple job processes. No consideration of execution time or initialization time is made.

Default Value SHRINK_BY_INVESTMENT

Type Tuning

ducc.rm.initialization.cap

The type of jobs supported by DUCC generally have very long and often fragile initialization periods. Errors in the applications and other problems such is missing or errant services can cause processes to fail during this phase.

To avoid preempting running jobs and allocating a large number of resources to jobs only to fail during initialization, the Resource Manager schedules a small number of processes until it is determined that the initialization phase will succeed.

The initialization.cap determines the maximum number of processes allocated to a job until at least one process successfully initializes. Once any process initializes the Resource Manager will proceed to allocate the job its full fair share of processes.

The initialization cap can be overridden on a class basis by configuration via ducc.classes.

Default Value 1

Type Tuning

ducc.rm.expand.by.doubling

When a job expands because its fair share has increased, or it has completed initialization, it may be desired to govern the rate of expansion. If expand.by.doubling is set to "true", rather than allocate the full fair share of processes, the number of processes is doubled each scheduling cycle, up to the maximum allowed.

Expand.by.doubling can be overridden on a class basis by configuration via ducc.classes.

Default Value true

Type Tuning

ducc.rm.prediction

Because initialization time may be very long, it may be the case that a job that might be eligible for expansion will be able to complete in the currently assigned shares before any new processes are able to complete their initialization. In this case expansion results in waste of resources and potential eviction of processes that need not be evicted.

The Resource Manager monitors the rate of task completion and attempts to predict the maximum number of processes that will be needed at a time in the future based on the known process initialization time. If it is determined that expansion is unnecessary then it is not done for the job.

Prediction can be overridden on a class basis by configuration via ducc.classes.

Default Value true

Type Tuning

ducc.rm.prediction.fudge

When ducc.rm.prediction is enabled, the known initialization time of a job's processes plus some "fudge" factor is used to predict the number of future resources needed. The "fudge" is specified in milliseconds.

The default "fudge" is very conservative. Experience and site policy should be used to set a more practical number.

Prediction.fudge can be overridden on a class basis by configuration via ducc.classes.

Default Value 120000

Type Tuning

ducc.rm.defragmentation.threshold

If *ducc.rm.defragmentation* is enable, limited defragmentation of resources is performed by the Resource Manager to create sufficient space to schedule work that has insufficient resources (new jobs, for example.). The

term *insufficient* is defined as “needing more processes than the defragmentation threshold, but currently having fewer processes than the defragmentation threshold.” These are called “needy” jobs. Additionally, the Resource Manager will never evict processes from “needy” jobs for the purpose of defragmentation.

This property allows installations to customize the value used to determine if a job is “needy”. Jobs with fewer processes than this are potentially needed, and jobs with more processes are never needy.

Default Value 8

Type Tuning

ducc.rm.admin.endpoint
This JMS endpoint used for RM administrative requests.

Default Value ducc.rm.admin.channe.

Type Private

ducc.rm.admin.type
This is the JMS endpoint type used for RM administrative requests.

Default Value ducc.rm.admin.channe.

Type Private

12.4.7 Agent Properties

ducc.agent.configuration.class
This is the name of the pluggable java class used to implement the DUCC Agents.

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

Type Private

ducc.agent.request.endpoint
This is the JMS endpoint through which agents receive state from the Process Manager.

Default Value ducc.agent

Type Private

ducc.agent.request.endpoint.type
This is the JMS endpoint type used for state messages sent by the Process Manager.

Default Value topic

Type Private

ducc.agent.managed.process.state.update.endpoint
This is the JMS endpoint used to communicate from the managed process to the Agent (Job Process).

Default Value ducc.managed.process. state.update

Type Private

ducc.agent.managed.process.state.update.endpoint.type
This is the JMS endpoint type used to communicate from the managed process (Job Process) to the Agent.

Default Value socket

Type Private

ducc.agent.managed.process. state.update.endpoint.params
These are configuration parameters for the Agent-to-JP communication socket. These should only be modified by DUCC developers.

Default Value transferExchange=true&sync=false

Type Private

ducc.agent.node.metrics.endpoint
This is the JMS endpoint used to send node metrics updates to listeners. Listeners are usually the Resource Manager and Web Server. These messages serve as node "heartbeats". As well, the node metrics heartbeats contain the amount of RAM on the node and the number of processors.

Default Value ducc.node.metrics

Type Private

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

Default Value topic

Type Private

ducc.agent.node.metrics.publish.rate
The interval in milliseconds between node metric publications. Every agent publishes its updates at this rate. On large clusters, a high rate (small interval) can be a burden on the network.

Note: the Resource Manager uses the data in the node metrics for scheduling.

Default Value 60000

Type Tuning

ducc.agent.node.inventory.endpoint
This is the JMS endpoint used to send node inventory messages to listeners. Listeners are usually the Orchestrator and Web Server. Information in these messages include a map of processes being managed on the node.

Default Value ducc.node.inventory

Type Private

ducc.agent.node.inventory.endpoint.type
This is the JMS endpoint type used to send node inventory updates from the agents.

Default Value topic

Type Private

ducc.agent.node.inventory.publish.rate
The interval in milliseconds between node inventory publications.

If the inventory has not changed since the last update the agent bypasses sending the update, up to a maximum of ducc.agent.node.inventory.publish.rate.skip times.

Default Value 10000

Type Tuning

ducc.agent.node.inventory.publish.rate.skip
This is the number of times the agent will bypass publishing its node inventory if the inventory has not changed.

Default Value 30

Type Tuning

ducc.agent.launcher.thread.pool.size
This is establishes the size of the agent's threadpool used to manage spawned processes.

Default Value 10

Type Tuning

ducc.agent.launcher.ducc_spawn_path

This property specifies the full path to the ducc_ling utility. During installation ducc_ling is normally moved to local disk and given setuid-root privileges. Use this property to tell the DUCC agents the location of the installed ducc_ling. The default location is within an architecture dependent subdiretory of DUCC_HOME/admin.

The arcitecture is derived from the JRE property *os.arch*. During DUCC installation the *ducc_ling* utility is compiled for the architecture of the host where DUCC is installed. In heterogeneous clusters, the system administrator should run the utility *build_duccling* once on a machine of each architecture to insure this utility gets correctly installed.

Default Value $DUCC_HOME/admin/$os.arch/ducc_ling

Type Tuning

ducc.agent.launcher.process.stop.timeout

This property specifies the time, in milliseconds, the agent should wait before forcibly terminating a job process (JP) after an attempted graceful shutdown. If the child process does not terminate in the specified time, it is forcibly terminated with kill -9.

This type of stop can occur because of preemption or system shutdown.

Default Value 60000

Type Tuning

ducc.agent.launcher.process.init.timeout

This property specifies the time, in milliseconds, that the agent should wait for a job process (JP) to complete initialization. If initialization is not completed in this time the process is terminated and and Initialization-Timout status is send to the job driver (JD) which decides whether to retry the process or terminate the job.

Default Value 7200000

Type Tuning

ducc.agent.share.size.fudge.factor

The DUCC agent monitors the size of the resident memory of its spawned processes. If a process exceeds its declared memory size by any significant amount it is terminated and a ShareSizeExceeded message is sent. The Job Driver counts this towards the maximum errors for the job and will eventually terminate the job if excessive such errors occur.

This property defines the percentage over the declared memory size that a process is allowed to grow to before being terminated.

To disable this feature, set the value to -1.

Default Value 5

Type Tuning

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

The DUCC Agents scan nodes for processes that should not be running; for example, a job may have left a 'rogue' process alive when it exits, or a user may log in to a node unexpectedly. These processes are reported to the administrators via the webserver for possible action.

This configuration parameter enumerates userids which are ignored by the rogue-process scan.

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

Type Tuning

ducc.agent.rogue.process.exclusion.filter

The DUCC Agents scan nodes for processes that should not be running; for example, a job may have left a 'rogue' process alive when it exits, or a user may log in to a node unexpectedly. These processes are reported to the administrators via the webserver for possible action.

This configuration parameter enumerates processes by name which are ignored by the rogue process detector.

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

Type Tuning

ducc.agent.launcher.cgroups.enable

Enable or disable CGroups support. If CGroups are not installed on a specific machine, this is ignored.

With CGroups the RSS for a managed process (plus any children processes it may spawn) is limited to the allocated share size. Additional memory use goes to swap space. DUCC monitors and limits swap use to the same proportion of total swap space as allocated share size is to total RAM. If a process exceeds its allowed swap space it is terminated and a ShareSizeExceeded message is sent to the Job Driver.

Nodes not using CGroups fall back to the ducc.agent.share.size.fudge.factor.

Default Value true

Type Tuning

ducc.agent.launcher.cgroups.utils.dir

Location of CGroups programs, like cgexec. If CGroups are not installed on a specific machine, this is ignored.

Depending on the OS, CGroups programs may be installed in different places. Provide a comma separated list of directories the agent should search to find the programs.

Default Value /usr/bin

Type Tuning

ducc.agent.exclusion.file

This specifies the exclusion file to enable node based exclusion for various features. Currently only CGroup exclusion is supported.

The exclusion file has one line per agent of the form:

```
<node>=cgroups
```

If the keyword "cgrouops" is found, the node is excluded from CGroup support.

Default Value Not configured.

Type Tuning

12.4.8 Process Manager Properties

ducc.pm.configuration.class

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

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

Type Private

ducc.pm.request.endpoint

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

Default Value ducc.pm

Type Private

ducc.pm.request.endpoint.type

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

Default Value queue

Type Private

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

Default Value ducc.pm.state

Type Private

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

Default Value topic

Type Private

ducc.pm.state.publish.rate
The interval in milliseconds between process manager heartbeat publications.

Default Value 25000

Type Private

12.4.9 Job Process Properties

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

Default Value org.apache.uima.ducc.transport.configuration.jp.JobProcessConfiguration

Type Private

ducc.uima-as.endpoint
This is the endpoint through which job processes (JPs) receive messages from the Agents.

Default Value ducc.job.managed.service

Type Private

ducc.uima-as.endpoint.type
This is the JMS endpoint type used for messages sent to the JPs from the Agents.

Default Value socket

Type Private

ducc.uima-as.endpoint.params
This configures the JP-to-Agent communication socket. It should be changed only by DUCC developers.

Default Value transferExchange=true&sync=false

Type Private

ducc.uima-as.saxon.jar.path
This configures the path the required Saxon jar.

Default Value file:$DUCC_HOME/apache-uima/saxon/saxon8.jar

Type Private

ducc.uima-as.dd2spring.xsl.path
This configures the path the required dd2spring xsl definitions.

Default Value $DUCC_HOME/apache-uima/bin/dd2spring.xsl

Type Private

ducc.uima-as.flow-controller.specifier
This configures the pluggable class that implements the default flow controller used in the DUCC job processes (JPs).

Default Value org.apache.uima.ducc.FlowController

Type Private

ducc.process.request.timeout
This is the maximum amount of time to wait for a response from a JD, in milliseconds. This value is used by a JP when sending requests to the JD.

Default Value 30000

Type Tuning

ducc.process.uima.as.container.class
Define process container class for DD jobs to instantiate and invoke via reflection. The container provides classpath isolation for user defined analytics. The container is instantiated with classes from a System classloader.

Default Value org.apache.uima.ducc.user.jp.UimaASProcessContainer

Type Private

ducc.process.uima.container.class
Define process container class for non-DD jobs to instantiate and invoke via reflection. The container provides classpath isolation for user defined analytics. The container is instantiated with classes from a System classloader.

Default Value org.apache.uima.ducc.user.jp.UimaProcessContainer

Type Private

ducc.process.thread.sleep.time
Define the sleep time in milliseconds for JP to use when JD sends empty CAS. In this case the JD's CR has processed its collection. The JP threads need to slow down sending requests

Default Value 3000

Type Tuning

12.5 ducc.private.properties

12.5.1 Web Server Properties

ducc.ws.port.ssl.pw
This is the password used to generate the Web Server's keystore used for HTTPS requests. Usually this keystore is created at initial installation time using ducc_post_install.

Default Value quackquack

Type Local

12.6 Resource Manager Configuration: Classes and Nodepools

The class configuration file is used by the Resource Manager configures the rules used for job scheduling. See the Resource Manager chapter for a detailed description of the DUCC scheduler, scheduling classes, and how classes are

used to configure the scheduling process.

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

12.6.1 Nodepools

Overview

A *nodepool* is a grouping of a subset of the physical nodes to allow differing scheduling policies to be applied to different nodes in the system. Some typical nodepool groupings might include:

1. Group Intel and Power nodes separately so that users may submit jobs that run only in Intel architecture, or only Power, or "don't care".
2. Designate a group of nodes with large locally attached disks such that users can run jobs that require those disks.
3. Designate a specific set of nodes with specialized hardware such as high-speed network, such that jobs can be scheduled to run only on those nodes.

A Nodepool is a subset of some larger collection of nodes. Nodepools themselves may be further subdivided. Nodepools may not overlap: every node belongs to exactly one nodepool. During system start-up the consistency of nodepool definition is checked and the system will refuse to start if the configuration is incorrect.

NOTE: The administrative command *check_ducc -c* may be used to verify and validate you class configration before attemping to start DUCC. *check_ducc -cv* may be used to additionally "pretty-print" the ducc.classes configuration to the console to revel class nesting and inheritance.

For example, the diagram below is an abstract representation of all the nodes in a system. There are five nodepools defined:

- Nodepool "NpAllOfThem" is subdivided into three pools, NP1, NP2, and NP3. All the nodes not contained in NP1, NP2, and NP3 belong to the pool called "NpAllOfThem".
- Nodepool NP1 is not further subdivided.
- Nodepool NP2 is not further subdivided.
- Nodepool NP3 is further subdivided to form NP4. All nodes within NP3 but not in NP4 are contained in NP3.
- Nodepool NP4 is not further subdivided.

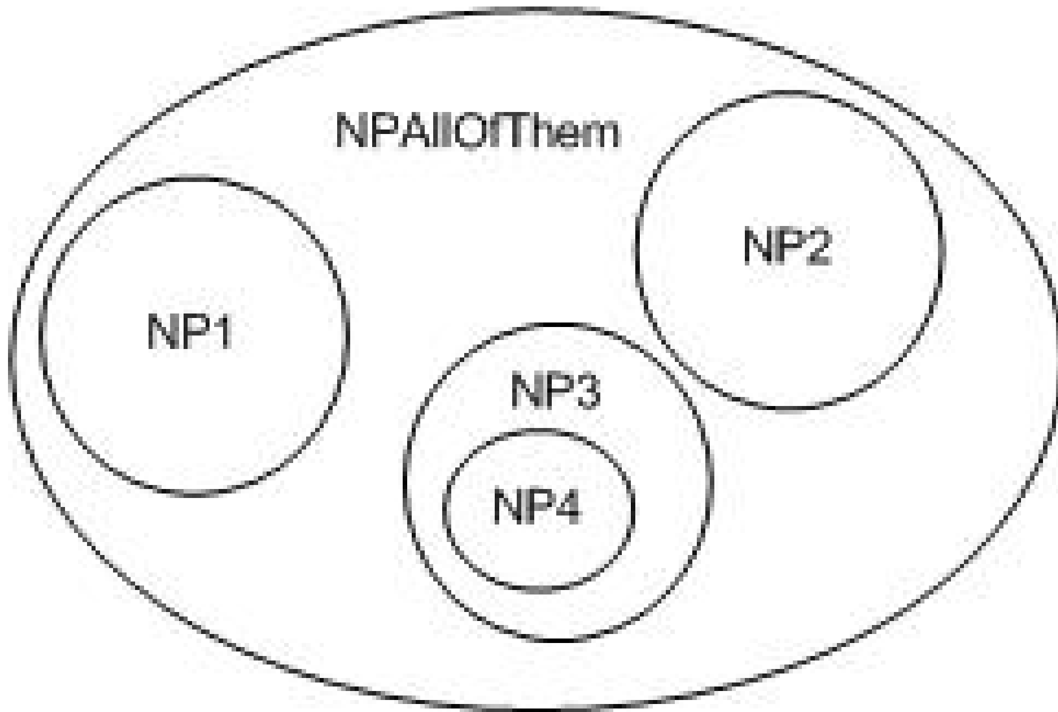

Figure 12.1: Nodepool Example

In the figure below the Nodepools are incorrectly defined for two reasons:

1. NP1 and NP2 overlap.
2. NP4 overlaps both nodepool "NpAllOfThem" and NP3.

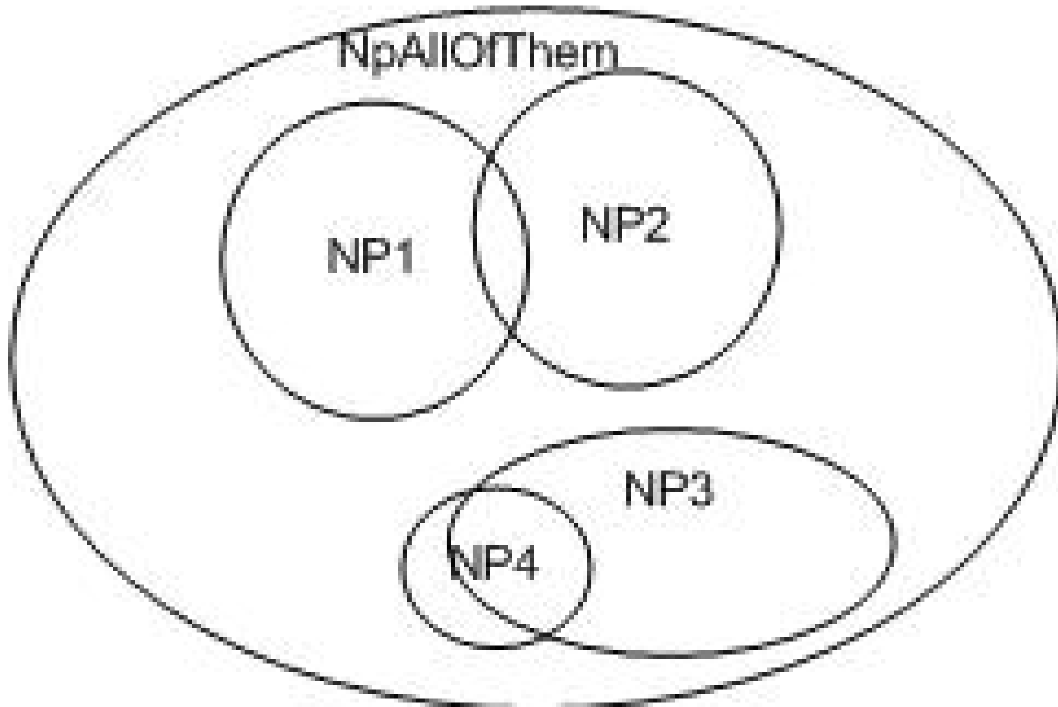

Figure 12.2: Nodepools: Overlapping Pools are Incorrect

Multiple "top-level" nodepools are allowed. A "top-level" nodepool has no containing pool. Multiple top-level pools logically divide a cluster of machines into *multiple independent clusters* from the standpoint of the scheduler. Work scheduled over one pool in no way affects work scheduled over the other pool. The figure below shows an abstract nodepool configuration with two top-level nodepools, "Top-NP1" and "Top-NP2".

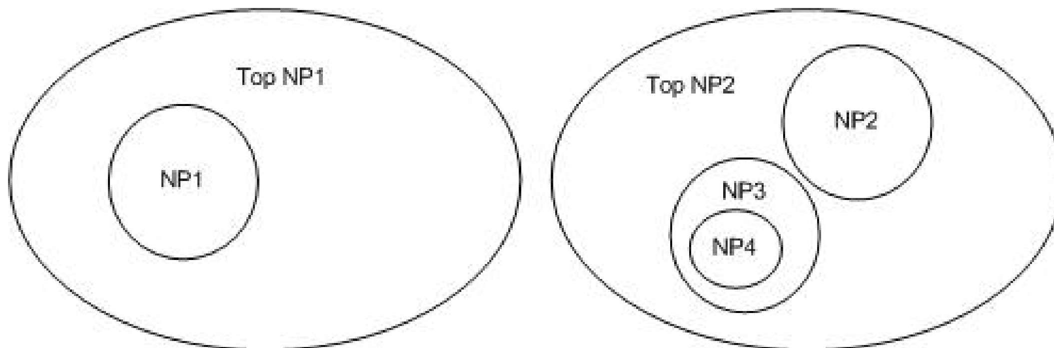

Figure 12.3: Nodepools: Multiple top-level Nodepools

Scheduling considerations

A primary goal of the scheduler is to insure that no resources are left idle if there is pending work that is able to use those resources. Therefore, work scheduled to a class defined over a specific nodepool (say, NpAllOfThem), may be scheduled on nodes in any of the nodepools contained within NpAllOfThem. If work defined over a subpool (such as NP1) arrives, processes on nodes in NP1 that were scheduled for NpAllOfThem are considered *squatters* and are the most likely candidates for eviction. (Processes assigned to their proper nodepools are considered *residents* and are evicted only after all *squatters* have been evicted.) The scheduler strives to avoid creating *squatters*.

Because non-preemptable allocations can't be preempted, work submitted to a class implementing one of the non-preemptable policies (FIXED or RESERVE) are never allowed to "squat" in other nodepools and are only scheduled on nodes in their proper nodepool.

In the case of multiple top-level nodepools: these nodepools and their sub-pools form independent scheduling groups. Specifically,

- Fair-share allocations over any nodepool in one top-level pool do NOT affect the fair-share allocations for jobs in any other top-level nodepool. Top-level nodepools define independently scheduled of resources within a single DUCC cluster.
- Work submitted to classes under one top-level nodepool do NOT get expanded to nodes under another top-level nodepool, even is there is sufficient capacity.

Most installations will want to assign the majority of nodes to a single top-level nodepool (or its subpools), using other top-level pools for nodes that cannot be shared with other work.

Configuration

DUCC uses simple named stanzas containing key/value pairs to configure nodepools.

At least one nodepool definition is required. This nodepool need not have any subpools or node definitions. The first top-level nodepool is considered the "default" nodepool. Any node not named specifically in one of the node files which checks in with DUCC is assigned to this first, *default* nodepool.

Thus, if only one nodepool is defined with no other attributes, all nodes are assigned to that pool.

A nodepool definition consists of the token "Nodepool" followed by the name of the nodepool, followed by a block delimited with "curly" braces { and }. This block contains the attributes of the nodepool as key/value pairs. Lineneds

are ignored. A semicolon ";" may optionally be used to delimit key/value pairs for readability, and an equals sign "=" may optionally be used to delimit keys from values, also just for readability. See the below.

The attributes of a Nodepool are:

domain This is valid only in the "default" (first) nodepool. Any node in any nodefile which does not have a domain, and any node which checks in to the Resource Manager without a domain name is assigned this domain name in order that the scheduler may deal entirely with full-qualified node names.

If no *domain* is specified, DUCC will attempt to guess the domain based on the domain name returned on the node where the Resource Manager resides.

nodefile This is the name of a file containing the names of the nodes which are members of this nodepool.

parent This is used to indicate which nodepool is the logical parent. Any nodepool without a *parent* is considered a top-level nodepool.

The following example defines six nodepools,

1. A top-level nodepool called "–default–". All nodes not named in any nodefile are assigned to this nodepool.
2. A top-level nodepool called "jobdriver", consisting of the nodes named in the file *jobdriver.nodes*.
3. A subpool of "–default–" called "intel", consisting of the nodes named in *intel.nodes*.
4. A subpool of "–default–" called "power", consisting of the nodes named in the file *power.nodes*.
5. A subpool of "intel" called "nightly-test", consisting of the nodes named in *nightly-test.nodes*.
6. And a subpool of "power" called "timing-p7", consisting of the nodes named in *timing-p7.nodes*.

```
Nodepool --default--  { domain mydomain.net }
Nodepool jobdriver    { nodefile jobdriver.nodes }

Nodepool intel        { nodefile intel.nodes        ; parent --default-- }
Nodepool power        { nodefile power.nodes        ; parent --default-- }

Nodepool nightly-test { nodefile nightly-test.nodes ; parent intel }
Nodepool timing-p7    { nodefile timing-p7.nodes    ; parent power }
```

Figure 12.4: Sample Nodepool Configuration

12.6.2 Class Definitions

Scheduler classes are defined in the same simple block language as nodepools.

A simple inheritance (or "template") scheme is supported for classes. Any class may be configured to "derive" from any other class. In this case, the child class acquires all the attributes of the parent class, any of which may be selectively overridden. Multiple inheritance is not supported but nested inheritance is; that is, class A may inherit from class B which inherits from class C and so on. In this way, generalized templates for the site's class structure may be defined.

The general form of a class definition consists of the keyword Class, followed by the name of the class, and then optionally by the name of a "parent" class whose characteristics it inherits. Following the name (and optionally parent class name) are the attributes of the class, also within a { block } as for nodepools, and with lines and key/value pairs optionally delimited by ";" and "=", respectively. See the sample below.

The attributes defined for classes are:

abstract If specified, this indicates this class is a template ONLY. It is used as a model for other classes. Values are "true" or "false". The default is "false". This class is never passed to the scheduler and may not be referenced by jobs.

debug FAIR_SHARE only. This specifies the name of a class to substitute for jobs submitted for debug. For example, if class *normal* specifies

```
debug = fixed
```

then any job submitted to this class with debugging requested is actually scheduled in class *fixed*. (For example, one probably does not want a debugging job scheduled as FAIR_SHARE and possibly preempted, preferring the non-preemptable class *fixed*.

default This specifies the class to be used as the default class for work submission if no class is explicitly given. Only one class of type FAIR_SHARE may contain this designation, in which case it names the default FAIR_SHARE class. Only one class of type FIXED_SHARE or RESERVE may contain this designation, in which case it names the default class to use for reservations (Note that either FIXED_SHARE or RESERVE scheduling policies are valid for reservations.)

expand-by-doubling FAIR_SHARE only. If "true", and the *initialization-cap* is set, then after any process has initialized, the job will expand to its maximum allowable shares by doubling in size each scheduling cycle.

If not specified, the global value set in ducc.properties is used.

initialization-cap FAIR_SHARE only. If specified, this is the largest number of processes this job may be assigned until at least one process has successfully completed initialization.

If not specified, the global value set in ducc.properties is used.

max-processes FAIR_SHARE and FIXED_SHARE only. This is the largest number of FIXED-SHARE, non-preemptable shares any single job may be assigned.

Omit this property, or set it to 0 to disable the cap.

g

prediction-fudge FAIR_SHARE only. When the scheduler is considering expanding the number of processes for a job it tries to determine if the job may complete before those processes are allocated and initialized. The *prediction-fudge* adds some amount of time (in milliseconds) to the projected completion time. This allows installations to prevent jobs from expanding when they were otherwise going to end in a few minutes anyway.

If not specified, the global value set in ducc.properties is used.

nodepool Jobs for this class are assigned to nodes in this nodepool. The value must be the name of one of the configured nodepools.

policy This is the scheduling policy, one of FAIR_SHARE, FIXED_SHARE, or RESERVE. This attribute is required (there is no default).

priority This is the scheduling priority for jobs in this class.

weight FAIR_SHARE only. This is the fair-share weight for jobs in this class.

The following figure illustrates a representative class configuration for a large cluster, consisting of mixed Intel and Power nodes. This class definition assumes the nodepool configuration shown above. FAIR_SHARE, FIXED_SHARE, and RESERVE classes are defined over each machine architecture, Intel and Power, and over the combined pool.

```
# --------------------- Fair share definitions ---------------
Class fair-base {
      policy = FAIR_SHARE
      nodepool = intel
      priority = 10
      weight = 100
      abstract = true
      debug = fixed
}

Class nightly-test   fair-base  { weight = 100; nodepool nightly-test; priority = 7}

Class background     fair-base  { weight = 20 }
Class low            fair-base  { weight = 50 }
Class normal         fair-base  { weight = 100; default = true }
Class high           fair-base  { weight = 200 }
Class weekly         fair-base  { weight = 400 }

Class background-p7  background { nodepool = power }
Class low-p7         low        { nodepool = power }
Class normal-p7      normal     { nodepool = power }
Class high-p7        high       { nodepool = power }
Class weekly-p7      weekly     { nodepool = power }

Class background-all background { nodepool = --default-- }
Class low-all        low        { nodepool = --default-- }
Class normal-all     normal     { nodepool = --default-- }
Class high-all       high       { nodepool = --default-- }
Class weekly-all     weekly     { nodepool = --default-- }

# --------------------- Fixed share definitions ---------------
Class fixed-base {
      policy = FIXED_SHARE
      nodepool = intel
      priority = 5
      abstract = true
      max-processes = 10
}

Class fixed     fixed-base { }
Class fixed-p7  fixed-base { nodepool = power;    default = true; }
Class JobDriver fixed-base { nodepool = jobdriver; priority = 0 }

# --------------------- Reserve definitions ---------------
Class reserve-base {
      policy = RESERVE
      nodepool = intel
      priority = 1
      abstract = true
}

Class reserve     reserve-base { }
Class reserve-p7  reserve-base { nodepool = power }
Class timing-p7   reserve-base { nodepool = timing-p7 }
```

Figure 12.5: Sample Class Configuration

12.6.3 Validation

The administrative command, *check_ducc* may be used to validate a configuration, with the *-c* and *v* options. This reads the entire configuration and nodefiles, validates consistency of the definitions and insures the nodepools do not overlap.

The *start_ducc* command always runs full validation, and if the configuration is found to be incorrect, the cluster is not started.

Configuration checking is done internally by the DUCC java utility *org.apache.uima.ducc.commonNodeConfiguration*. This utility contains a public API as described in the Javadoc. It may be invoked from the command line as follows:

Usage:

```
java org.apache.uima.ducc.commonNodeConfiguration [-p] [-v nodefile] configfile
```

Options:

$-p$ Pretty-print the compiled configuration to stdout. This illustrates nodepool nesting, and shows the fully-completed scheduling classes after inheritance.

$-v$ **nodefile** This should be the master nodelist used to start DUCC. This is assumed to be constructed to reflect the nodepool organization as described here. If provided, the nodepools are validated and checked for overlaps.

configfile This is the name of the file containing the configuration.

12.7 Ducc Node Definitions

The DUCC node definitions are specified by default in the file *ducc.nodes*.

The DUCC node list is used to configure the nodes used to run jobs and assign reservations. When DUCC is started, the nodelist is read an a DUCC Agent is started on every node in the list.

The node list can be composed of multiple node lists to assist organization of the DUCC cluster. All the administrative commands operate upon node lists. By carefully organized these lists it is possible to administer portions of a cluster independently.

In particular, it is highly recommended that the nodelists reflect the nodepool structure. In this way, the configuration used to start DUCC is guaranteed to match the nodeppool definitions.

Several types of records are permitted in nodelists:

Comments A comment starts with the symbol "#". All text on the line following this symbol is ignored.

import If a line starts with the symbol *import*, the next symbol on that line is expected to be the name of another node list. This permits the DUCC cluster's nodes to be configured in a structured manner.

For instance, the file *ducc.nodes* might consist entirely of *import* statements naming all of the nodepool files.

domain This must be the first line of the file. If specified, it should name the default domain to be used for all the nodes in this file, and the nodes named in imported files. If not specified, then during start-up, nodes without domain names are assigned domain names according to the global domain name specified in the Resource Manager configuration file, and if none is specified there, the domain name on the host starting DUCC is used.

nodename This is a single token consisting of the name of a node on which an agent it to be started.

The example below shows a partial, hypothetical node configuration corresponding to the nodepool configuration above.

```
> cat ducc.nodes
# import all the nodes corresponding to my nodepools
domain my.domain
import intel.nodes
import power.nodes
import jobdriver.nodes
import nightly-test.nodes
import timing-p7.nodes

> cat intel.nodes
# import the intel nodes, by frame
import intel-frame1.nodes
import intel-frame2.nodes
import intel-frame3.nodes

>cat intel-frame1.nodes
#import the specific nodes from frame1
r1s1node1
r1s1node2
r1s1node3
r1s1node4
```

Figure 12.6: Sample Node Configuration

12.8 Ducc User Definitions

The DUCC user registry provides user-specific overrides of various constraints DUCC might impose.

As of 2.0.0, the only constraint override is allotment for non-preemptable requests.

The syntax of the user registry is the same as that used in *ducc.classes*, and in fact, the user registry may be embedded directly in that file, rather than specified externally.

The registry consists of multiple entries, one for affected user. Any user of the system NOT in the registry acquires the system defaults.

A user definition consists of the token "User" followed by the id of the user, followed by a block delimited with "curly" braces { and }. This block contains the attributes of the nodepool as key/value pairs. Lineneds are ignored. A semicolon ";" may optionally be used to delimit key/value pairs for readability, and an equals sign "=" may optionally be used to delimit keys from values, also just for readability.

The attributes of a User entry are:

max-allotment This overrides the maximum allotment for non-preemptable requests as defined in *ducc.properties*. The override may be used to either increase, or decrease the user's allotment. The units are in gigabytes.

The example below shows overrides for three users:

- Bob is allowed a non-preemptable allotment of 4000GB.
- May is allowed a non-preemptable allotment of 1000GB.
- Antoinette is allowed a non-preemptable allotment of 720GB.

```
# --------------------- User Registry ---------------------
User bob         { max-allotment = 4000 }
User mary        { max-allotment = 1000 }
User antoinette  { max-allotment = 720  }
```

Figure 12.7: Sample User Registration

12.9 Administrative Commands

The administrative commands include a command to start DUCC, one to stop it, and one to verify the configuration and query the state of the cluster.

Note: The scripting that supports these functions runs (by default) in multi-threaded mode so large clusters can be started, stopped, and queried quickly. If DUCC is running on an older system, the threading may not work right, in which case the scripts detect this and run single-threaded. As well, all these commands support a "–nothreading" option to manually disable the threading.

12.9.1 start_ducc

Description

The command $DUCC_HOME/admin/start_ducc is used to start DUCC processes. If run with no parameters it takes the following actions:

- Starts the management processes Resource Manager, Orchestrator, Process Manager, Services Manager, and Web Server on the local node (where start_ducc is executed.
- Starts an agent process on every node named in the default node list.

Usage

start_ducc [options]
If no options are given, all DUCC processes are started, using the default node list, *ducc.nodes*.

Options:

-n, –nodelist [nodefile]
Start agents on the nodes in the nodefile. Multiple nodefiles may be specified:

```
start\_ducc -n foo.nodes -n bar.nodes -n baz.nodes
```

-c, –component [component]
Start a specific DUCC component, optionally on a specific node. If the component name is qualified with a nodename, the component is started on that node. To qualify a component name with a destination node, use the notation component@nodename. Multiple components may be specified:

```
start\_ducc -c sm -c pm -c rm -c or@bj22 -c agent@n1 -c agent@n2
```

Components include:

rm The Resource Manager

or The Orchestrator

pm The Process Manager

sm The Service Manager

ws The Web Server

agent Node Agents

–nothreading If specified, the command does not run in multi-threaded mode even if it is supported on the local platform.

Notes:

A different nodelist may be used to specify where Agents are started. As well multiple node lists may be specified, in which case Agents are started on all the nodes in the multiple node lists.

To start only agents, run start_ducc specifying a nodelist explicitly. When started like this, the management daemons are not started unless explicitly requested.

To start only management processes, run start_ducc with the -m or –management flags. When started like the the agents are not started unless explicitly requested.

To start a specific management process, run start_ducc with the -c component parameter, specify the component that should be started.

Examples:

Start some nodes from two different nodelist. This doesn't start any of the management processes but it does insure the ActiveMQ Broker is available.

```
start\_ducc -n foo.nodes -n bar.nodes
```

Start an agent on a specific node:

```
start\_ducc -c agent@a.specific.node
```

Start the webserver on node 'bingle':

```
start\_ducc -c ws@bingle
```

Debugging:

Sometimes something will not start and it can be difficult to understand why. To diagnose, it is helpful to know that *start_ducc* is simply a wrapper around a lower-level bit of scripting that does the actual work. That lower-level code can be invoked stand-alone, in which case console messages that *check_ducc* will have suppressed are presented to the console.

The lower-level script is called *ducc.py* and accepts the same *-c component* flag as start_ducc. If some component will not start, try running *ducc.py -c component* directly. It will start in the foreground and usually the cause of the problem becomes evident from the console.

For example, suppose the Resource Manager will not start. Run the following:

```
./ducc.py -c rm
```

and examine the output. Use *CTL-C* to stop the component when done.

12.9.2 stop_ducc

Description:

Stop_ducc is used to stop DUCC processes. If run with no parameters it takes the following actions: TODO: Garbled by maven or docbook, update this

Usage:

stop_ducc [options]
If no options are given, help text is presented. At least one option is required, to avoid accidental cluster shutdown.

Options:

-a –all
Stop all the DUCC processes, including agents and management processes. This broadcasts a "shutdown" command to all DUCC processes. Shutdown is normally performed gracefully will all process including job processes given time to save state. All user processes, both jobs and services, are sent shutdown signals. Job and service processes which do not shutdown within a designated grace period are then forcibly terminated with kill -9.

-n, –nodelist [nodefile]
Only the DUCC agents in the designated nodelists are shutdown. The processes are sent kill -INT signals which triggers the Java shutdown hooks and enables graceful shutdown. All user processes on the indicated nodes, both jobs and services, are sent "shutdown" signals and are given a minute to shutdown gracefully. Job and service processes which do not shutdown within a designated grace period are then forcibly terminated with kill -9.

```
stop\_ducc -n foo.nodes -n bar.nodes -n baz.nodes
```

-c, –component [component]
Stop a specific DUCC component.

This may be used to stop an errant management component and subsequently restart it (with start_ducc).

This may also be used to stop a specific agent and the job and services processes it is managing, without the need to specify a nodelist.

Examples:

Stop agents on nodes n1 and n2:

```
stop_ducc -c agent@n1 -c agent@n2
```

Stop and restart the rm:

```
stop_ducc -c rm
start_ducc -c rm
```

Components include:

rm The Resource Manager.

or The Orchestrator.

pm The Process Manager.

sm The Service Manager.

ws The Web Server.

broker The ActiveMQ broker (only if the broker is auto-managed).

agentnode Node Agent on the specified node.

–nothreading If specified, the command does not run in multi-threaded mode even if it is supported on the local platform.

Notes:

Sometimes problems in the network or elsewhere prevent the DUCC components from stopping properly. The *check_ducc* command, described in the following section, contains options to query the existance of DUCC processes in the cluster, to forcibly (*kill -9*) terminate them, and to more gracefully terminate them (*kill -INT*).

12.9.3 check_ducc

Description:

Check_ducc is used to verify the integrity of the DUCC installation and to find and report on DUCC processes. It identifies processes owned by ducc (management processes, agents, and job processes), and processes started by DUCC on behalf of users.

Check_ducc can also be used to clean up errant DUCC processes when stop_ducc is unable to do so. The difference is that stop_ducc generally tries more gracefully stop processes. check_ducc is used as a last resort, or if a fast but graceless shutdown is desired.

Usage:

check_ducc [options] If no options are given this is the equivalent of:

```
check_ducc -c -n ../resources/ducc.nodes
```

This verifies the integrity of the DUCC installation and searches for all the processes owned by user *ducc* and started by DUCC on all the nodes in ducc.nodes.

Options:

-n –nodelist [nodefile] Only the nodes specified in the nodefile are searched. The option may be specified multiple times for multiple nodefiles. Note that the "local" node is always checked as well.

```
check_ducc -n nlist1 -n nlist2
```

-c –configuration Verify the Resource Manager configuration.

-p –pids Rewrite the PID file. The PID file contains the process ids of all known DUCC management and agent processes. The PID file is normally managed by start_ducc and stop_ducc and is stored in the file *ducc.pids* in directory *ducc_runtime/state*.

Occasionally the PID file can become partially or fully corrupted; for example, if a DUCC process dies spontaneously. Use check_ducc -p to search the cluster for processes and refresh the PID file.

-i, –int
Use this to send a shutdown signal (*kill -INT*) to all the DUCC processes. The DUCC processes catch this signal, close their resources and exit. Some resources take some time to close, or in case of problems, are unable to close, in which case the DUCC processes will unconditionally exit.

Sometimes problems in the network or elsewhere prevent *check_ducc -i* from terminating the DUCC processes. In this case, use *check_ducc -k*, described below.

-k, –kill
Use this to forcibly kill a component using kill -9. This should only be used if *stop_ducc* or *check_ducc -i* does not work.

-s, –single_user
Bypass the multi-user permission checks normally done on the ducc_ling utility.

–nothreading If specified, the command does not run in multi-threaded mode even if it is supported on the local platform.

-v, –verboase
When specified with *-c* to check the configuration, this emits a formatted version of the node list showing the full structure of the scheduling classes.

12.9.4 rm_reconfigure

Description:

Rm_reconfigure is used to force the Resource Manager (RM) to reread all its configuration files and reconfigure itself accordingly, without the need to fully stop and restart RM. This is generally much faster than RM restart and avoids losing most state messages from the other DUCC processes.

The *rm_reconfigure* command first performs a properties merge.

RM then validates the new configuration, and if no errors are found, saves certain information such as current node online-offline status. It then rereads the following configuration files and rebuilds its internal structures accordingly:

- ducc.properties (after merging default.ducc.properties and site.ducc.properties,
- ducc.classes,
- log4j.xml.

The saved configuration is then restored into the newly configured structures. On receipt of the next Orchestrator state, the RM fully rebuilds its state from the current DUCC load and scheduling restarts.

Depending on the nature of the new configuration, the current load may be adjusted; for example, if the weight of a fair-share class is changed, preemptions or extra allocations may be performed.

If the new configuration is not consistent with the current load, a number of more drastic adjustments will be performed:

- If a fair-share class is deleted, all existing jobs for that class are preempted and a *refusal* message is sent to the Orchestrator for each affected job.
- If a fair-share class is redefined over a different nodepool such that existing work are no longer legally scheduled, any shares allocated over inappropriate hosts are *preempted.* As soon as the preemptions are acknowledged, the RM will reschedule the shares over the differently-configured resources.
- If a non-preemptable class is deleted or reconfigured so existing non-preempt able work is no longer allocated correctly, the following will occur:
  - If the shares are for services, they are deallocated and a *refusal* is sent to the Orchestrator. The Service Manager will observe this and restart the processes, causing them to be reallocated over the changed configuration.
  - Otherwise, the RM leaves the allocation in place, but places the hosts on an internal *blacklist*, preventing subsequent scheduling to those hosts. Once the (now) incorrectly placed shares are freed (e.g. by canceling a reservation or exit of a managed reservation), the hosts are again white listed and made available for scheduling.

In short, the RM makes every effort to avoid disturbing existing allocations, and blacklists hosts that are no longer consistently configured for the current load, until such time as the allocations on those hosts are released.

Usage:

rm_reconfigure
This command has no options.

12.9.5 rm_qload

Description:

Rm_qload is used to query the Resource Manager's scheduling tables to determine the current demand and capacity of the system, as the RM sees it. The primary purpose is to provide information to adjunct resource managers

(such as a "cloud") to determine the current needs, or lack thereof, of the system. The administrative command is implemented as a Python script that interacts with the underlying Java "RmQueryLoadReply" API and is provided mostly as an example of how scripting can be used to interact with the RM.

After displaying the current scheduling quantum, the response is provided in two sections:

1. Information showing the current demand and usage of resource classes, and
2. Information showing the current nodepool usage.

Class section

Three lines are emitted per class:

1. The name of the class and its scheduling policy,
2. A line showing the *demand*, or *request* by quantum, on the class, and
3. A line showing the *usage*, or *award*, by quantum on the class.

The numbers shown for *request* and *award* show the number of processes, by memory, in terms of scheduling quantum, for each class. For example, assuming the scheduling quantum is 15GB, the following shows:

- Five processes of quantum 2 (15-30GB) are requested, but only two have been awarded,
- Three processes of quantum 3 (31-45GB) are requested and all have been awarded,
- Four processes of quantum 4 (46-60GB) are requested, and two have been awarded.

```
Class normal policy FAIR_SHARE
   requested    0    0    5    3    4    0    0    0    0
   awarded      0    0    2    3    2    0    0    0    0
```

Nodepool section

Six lines are displayed for each nodepool:

1. The name of the nodepool,
2. A summary showing the number hosts in the pool which are online, dead (unresponsive), and varied-off, the total quantum shares available to the nodepool, and the total unscheduled or *free* shares.
3. The number of hosts known to the nodepool, by quantum, similar to the class listings above,
4. The nubmer of online hosts, by quantum,
5. The number of completely free hosts by quantum (no work currently scheduled), and
6. The number of *virtual* hosts, by quantum. A *virtual host* is created when a host is partially scheduled. For example, if a 32G processes is scheduled on a 64G host, this creates one free 32G *virtual host*.

To determine the number of processes, by quantum, that can be scheduled, one must *sum* the "free" and "virtual" columns.

For example, (assuming a 15GB quantum), the following listing shows that nodepool "power" contains fourteen hosts with at least 45GB each (3 quanta). Two of these hosts have something scheduled on them (the "free machines" line), leaving unused space of one 15G quantum on one host, and one 30GB quantum on another host.

```
Nodepool power
   online 14 dead 0 offline 0 total-shares 42 free-shares 42
   all     machines:    0    0    0   14    0    0    0    0    0
   online  machines:    0    0    0    0    0    0    0    0    0
   free    machines:    0    0    0   12    0    0    0    0    0
   virtual machines:    0    1    1    0    0    0    0    0    0
```

Usage:

rm_qload
This command has no options.

12.9.6 rm_qoccupancy

Description:

Rm_qoccupancy provides a list of all known hosts to the RM, and for each host, the following information:

- The name of the host,
- Whether the host has any blacklists processes on it,
- Whether the host is currently onlline (responsive),
- The status of the host; whether the host is schedulable (*up* or *down*. A responsive host becomes unschedulable (*down*) if it is varied-off,
- The nodepool the host is a member of,
- The reported memory size of the host,
- The *order* of the host. The *order* is defined to be the maximum number of quantum shares supported by the host,
- The number of unscheduled quantum shares on the host, and
- If work is scheduled on the host, information relevent to that scheduled processes (or reservation).

If work is scheduled on a host, the work summary is keyed thus:

J The Orchestrator-assigned job id of the work,

S The RM-assigned share id of the work,

O The *order* of the allocation; that is, the number of quantum shares the allocation occupies,

II The *initialization investment*; the number of milliseconds the allocated work spent in its initialization phase, if any (usually only UIMA-AS processes display this),

IR The *runtime investment*; the number of milliseconds spent processing the current CASs, if this is a UIMA-AS processes. Note that this number can change dramatically, as it is the sum of time spent only by the current CASs. When a CAS completes, it no longer contributes to the investment of the process. The RM uses this information to determine the best candidate for eviction, if needed ot maintain fair-share.

E Whether the RM has preempted (evicted) the process but it has not yet exited,

P Whether the RM has purged the process (evicted, because the host is non-responsive), but it has not been confirmed evicted,

F Whether the process is *fixed*; that is, non-preemptbable,

I Whether the initialization phase is completed (usually only UIMA-AS processes).

The following example shows seven hosts, one with a preemptable share in the *–default–* nodepool (on bluej290-5), and one with a non-preemptable share in the *jobdriver* nodepool.

```
       Node Blacklisted Online Status        Nodepool     Memory Order   Free
  bluej290-5       False   True     up     --default--   32505856     2      0
           J[    6006] S[     189] O[2] II[        0] IR[        0] E[False] P[False] F[False] I[False]

  bluej290-6       False   True     up     --default--   32505856     2      2
  bluej290-7       False   True     up     --default--   32505856     2      2
```

```
bluej291-26       False   True     up    nightly-test   32505856     2      2
bluej291-27       False   True     up    nightly-test   32505856     2      2
bluej293-60       False   True     up           intel   32505856     2      2
bluej537-73       False   True     up       jobdriver   32505856     2      1
          J[    5973] S[       1] O[1] II[       0] IR[       0] E[False] P[False] F[ True] I[False]
```

Usage:

rm_qoccupancy
This command has no options.

12.9.7 vary_off

Description:

Vary_off is used to remove a host from scheduling and to evict the work that is running on it. This allows for graceful clearance of a host so the host can be take offline for maintenance, or any other purpose (such as sharing the host with other applications.) The DUCC agent is NOT stoppped; use stop_ducc to stop the agent. Reservations are not canceled by *vary_off*.

Only the userid that started DUCC may issue *vary_off*; attempts from other userids are rejected.

Usage:

vary_off list-of-hosts The *list-of-nodes* is a space delimited list of host to be removed from scheduling in the DUCC cluster.

12.9.8 vary_on

Description:

Vary_on is used to restore a host to scheduling by DUCC. If the agent is still alive the host becomes immediately available. The agent is not started by *vary_on*; use use start_ducc to start the agent if needed.

Only the userid that started DUCC may issue *vary_on*; attempts from other userids are rejected.

Usage:

vary_on list-of-hosts The *list-of-nodes* is a space delimited list of host to be restored for scheduling in the DUCC cluster.

12.9.9 ducc_properties_manager

Description: This CLI is used to manually merge or difference two properties files.

Normally, the DUCC scripts *start_ducc, check_ducc,*, and *rm_configure* automatically merge the file *default.ducc.properties* and *site.ducc.properties* when invoked.

Usage:

ducc_props_manager –merge file1 –with file2 –to file3 Merge two properties files into one. Properties added to, or changed in, the second file are used to override those in the first file, with the result written to the third file.

ducc_props_manager –delta file1 –with file2 –to file3 Compare two properties files and write the differences into a third file. The first file is considered a "master" file. Properties with different values in the second file, or which do not occur in the first file, are written into the third file.

Options:

−−merge file1 In this form, the two files specified in the −−*with* and −−*to* fields are merged, with the results placed in −−file3. Overrides are flagged with a change tag and the date of the merge.

file1 is considered the "master" properties file and is usually the unmodified file provided with the DUCC distribution, *default.ducc.properties.*

file2 is considered a set of override or additional properties and is usually the site local properties file, *site.ducc.properties.*

−−delta file1 In this form, the two files specified in the −−*with* and −−*to* fields are compared, with differences placed in −−file3.

file1 is considered the "master" properties file and is usually the unmodified file provided with the DUCC distribution, *default.ducc.properties.*

file2 is considered the "external" properties file and is usually the properties file from an older version of DUCC.

Differences are placed in −−*file3* which may be a viable first cut at a new *site.ducc.properties.*

−−with file2 This specifies the properties file to merge with the master, or to difference with the master properties file.

−−to file3 This specifies the file to which the results of the merge or delta are written.

Notes: None.

Chapter 13

Resource Management

13.1 Overview

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

Managed Jobs Managed jobs are Java applications implemented in the UIMA framework and are scaled out by DUCC as some number of discrete processes. Processes which compose managed jobs are always restartable and usually preemptable. Preemption occurs as a consequence of enforcing fair-share scheduling policies.

Services Services are long-running processes which perform some (common) function on behalf of jobs or other services. Services are scaled out as a set of, from the RM point of view, unrelated non-preemptable processes.

Reservations A reservation provides non-preemptable, persistent, dedicated use of a full machine or some part of a machine to a specific user.

Arbitrary Processes An *arbitrary process* or *managed reservation* is any process at all, which may or may not have anything to do with UIMA. These processes are typically used to run non-UIMA tasks such as application builds, large Eclipse workspaces for debugging, etc. These processes are usually scheduled as non-preemptable allocations, occupying either a dedicated machine or some portion of a machine.

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 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.

To compute the memory requirements for a job, the declared memory is rounded up to the nearest multiple of the share quantum. The total number of quantum shares for the job is calculated, and then divided by the number of quantum shares declared for the job to arrive at the number of processes to allocate. 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.

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.

Some work may be deemed to be more "important" than other work. To accommodate this, the RM implements a weighted fair-share scheduler. 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.

The abstraction used to organized jobs by fair-share weight is the job class or simply *class*. All job submissions must included a declared job class; if none is declared, a default class is chosen by DUCC. As jobs enter the system they are grouped with other jobs of the same class weight. The class abstraction and its attributes are described in subsequent sections.

The scheduler executes in three primary phases:

1. The How-Much phase: every job is assigned some number of quantum 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 for all jobs scheduled to a machine do not exceed the physical memory on the machine.
3. Defragmentation. If the what-of phase cannot allocate space according to the output of the how-much phase, the system is said to be *fragmented.* The RM scans for "rich" jobs and will attempt to preempt some small number of processes sufficient to guarantee every job gets at least one process allocation. (Note that sometimes this is not possible, in which case unscheduled work remaims pending until such time as space is freed-up.)

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. This phase takes into account all users and all jobs in the system.
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.
3. Job counts: For each user, collect all jobs assigned to that user and equally divide all the user's shares among the jobs. This apportions all shares given to this user for each class among the user's jobs in that class.

All non-preemptable allocations are restricted to one allocation per request. If space is available, the request succeeds immediately. If space can be made for the request through preemptions, the preemptions are scheduled and the reservation is deferred until space is available. If space cannot be found by means of preemption, the reservation remains pending until it either succeeds (by cancelation of other non-preemptive work, by adding resources to the system, or by increasing the user's non-preemptive allotment), or until it is canceled by the user or an administrator.

13.2 Preemption vs Eviction

The RM makes a subtle distinction between *preemption* and *eviction.*

Preemption occurs only as a result of fair-share calculations or defragmentation. Preemption is the process of deallocating shares from jobs beloing to users whose current allocation exceeds their fair-share, and conversely, only processes belonging to fair-share jobs can be preempted. This is generally dynamic: more jobs in the system result in a smaller fair-share for any given user, and fewer jobs result in a higher fair-share allocation.

Eviction occurs only as a result of system-detected errors, changes in node configuration, or changes in class configuration. Eviction may affect both preemptable work and some types of non-preemptable work.

Work that is non-preemptable, but restartable can be evicted. Such work consists of service processes (which are automatically resubmitted by the Service Manager), and managed reservations, which can be resubmitted by the user.

Unmanaged reservations are never evicted for any reason. If something occurs that would result in the reservation being (fatally) misplaced, the node is marked unschedulable and remains as such until the condition is corrected

or the reservation is canceled. Once the condition is repaired (either the reservaiton is canceled, or the problem is corrected), the node becomes schedulable again.

13.3 Scheduling Policies

The Resource Manager implements three scheduling policies. Scheduling policies are associated with *classes*.

FAIR_SHARE This is weighted-fair-share. All processes scheduled under fair-share are always *preemptable*.

FIXED_SHARE The FIXED_SHARE policy is used to allocate non-preemptable shares. The shares might be *evicted* as described above, but they are never *preempted*. Fixed share alloations are restricted to one allocation per request and may be subject to allotment caps.

FIXED_SHARE allocations have several uses:

- Unmaged reservations. In this case DUCC starts no work in the share(s); the user must log in (or run something via ssh), and then manually release the reservation to free the resources. This is often used for testing and debugging.
- Services. If a service is registered to run in a FIXED_SHARE allocation, DUCC allocates the resources, starts and manages the service, and releases the resource if the service is stopped or unregistered.
- UIMA jobs. A "normal" UIMA job may be submitted to a FIXED_SHARE class. In this case, the processes are never preempted, allowing constant and predictable execution of the job. The resources are automatically released when the job exits.
- Managed reservations. The *viaducc* utility is provided as a convenience for running managed reservations.

RESERVE The RESERVE policy is used to allocate a full, dedicated machine. The allocation may be *evicted* but it is never *preempted*. It is restricted to a single machine per request. The memory size specified in the reservation must match machine size exactly, within the limits of rounding to the next highest multiple of the quantum. DUCC will not "promote" a reservation request to a larger machine than is asked for. A reservation that does not adequately match any machine remains pending until resources are made available or it is canceled by the user or an administrator. Reservations may be subject to allotment caps.

13.4 Allotment

Allotment is a new concept introduced with DUCC 2.0.0 to prevent non-preemptable requests from dominating a cluster. This replaces the DUCC version 1 class policies of max-processes and max-machines.

It is possible to associate a maximum share allotment with any non-preemptable class. Allotment is assigned per user and is global across all non-preemptable classes. It is configured ducc.properties with *ducc.rm.global_allotment*.

A simple user registry provides per-user overrides of the global allotment as needed. The registry may be included in the class definition file (specified in ducc.properties under ducc.rm.class.definitions), or in a separate file, specified in ducc.properties as *ducc.rm.user.registry*.

13.5 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. Class-based priorities are used to resolve these conflicts.

Simply: priority is used to specify the order of evaluation of the job classes. Weight is used to proportionally allocate the number of shares to all classes of the same priority under the weighted fair-share policies.

Priority. When a scheduling cycle starts, the scheduling classes are ordered from "best" to "worst" priority. The scheduler then attempts to allocate ALL of the system's resources to the "best" priority class. If any resources are left, the scheduler proceeds to schedule classes in the next best priority, and so on, until either all the resources are exhausted or there is no more work to schedule.

It is possible to have multiple job classes of the same priority. What this means is that resources are allocated for the set of job classes from the same set of resources at the same time, usually under weighted fair-share. (It would be unusual to have multiple non-preemptable classes at the same priority. If this is configured, the class requests are filled arbitrarily with no attempt to divide the resources fairly or equitably). 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, fair-share weights are used for FAIR_SHARE classes, and allotment is used for non-preemptable classes.

Weight. Weight is used to determine the relative importance of jobs in a set of job classes of the same priority when doing fair-share allocation. All job classes of the same priority are assigned shares from the full set of available resources according to their weights using weighted fair-share. Weights are used only for fair-share allocation.

13.6 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.

Job classes are associated with nodepools. The scheduler treates preemptable work and non-preemptable work differently with regards to nodepools:

Preemptable work. The scheduler will attempt to allocate preemptable work in the nodepool associated with the work's class. If this nodepool becomes exhausted, and there are subpools, the scheduler proceeds to try to allocate resources within the subpools, recursively, until either all work is scheduled or there is no more work to schedule. (Allocations made within subpools are referred to as "squatters"; aloocations made in the directly associated nodepool are referred to as "residents".)

During eviction, the scheduler attempts to evict squatters first and only evicts residents once all the squatters are gone.

Non-Preemptable work. Non-preemptable work can only be allocated directly in the nodepool associated with the work's class. Such work can never become a squatter. The reason is that non-preemptbable squatters cannot be evicted, and so could dominate pools intended for other work.

More information on nodepools and their configuration can be found here.

13.7 Scheduling Classes

The primary abstraction to control and configure the scheduler is the *class*. A class is simply a set of rules used to parametrize how resources are assigned to work requests. Every request that enters the system is associated with a single class.

The scheduling 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 is used for 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.

Nodepool A class may be associated with exactly one nodepool. Fair-share jobs submitted to the class are assigned only resources which lie in that nodepool, or in any of the subpools defined within that nodepool. Non-preemptable requests must always be fulfilled from the nodepool assigned to the class; subpools are exempt from non-preemptable requests submitted to higher-level nodepools.

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 analytics 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 actual allocation 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. This avoids wasteful preemption of work to make space for other work that will be completing very soon anyway.

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.

If expand-by-doubling is disabled, jobs are allocated their full fair-share immediately.

More information on nodepools and their configuration can be found here.

Chapter 14

Service Management

The only administrative task relating to Service Management is registering global pingers, as described in *Service Pingers section* of this document.

A globally-registered service pinger is a properties file that contains only service registraton options pertaining to pingers. This file must be placed in DUCC's *runtime*/resources/service_monitors directory. It may be given any name but "best practices" would suggest it be named the same as the *service_ping_class*. Services then use this pinger by specifying its filename in their *service_ping_class* option.

Globally-registered pingers may be run internally as threads within the SM, or externally as processes. To specify that a pinger be run internally, add the property

```
internal = true
```

to the registration file. To specify that it run externall, add the property

```
internal = false
```

to the registration file.

The "internal" option is flagged as in illegal option when specified in service registrations and all pingers not specified as "internal" are run as *external* processes managed by the SM.

Best practices dictate that the filename of an *external* pinger contain the postfix *.external* to clearly identify it as external.

As an example, the default UIMA-AS pinger is supplied in the global registery under the two names:

```
org.apache.uima.ducc.cli.UimaAsPing
org.apache.uima.ducc.cli.UimaAsPing.external
```

Note that users may override any of the properties in globally-registered *external* pingers, but only the *service_ping_arguments* of an *internal* pinger to protect its integrity by speicfy that argument in their own service registrations.

Chapter 15

Simulation and System Testing

This chapter describes the large-scale testing and cluster-simulation tools supplied with DUCC. This is of use mostly to contributors and developers of DUCC itself.

DUCC is shipped with support for simulating large clusters of arbitrarily configured nodes. A simple control file describes some number of simulated nodes of arbitrary memory sizes. DUCC's design allows multiples of these to be spawned on a single node, or on a small set of nodes with multiple simulated nodes apiece. The standard testing configuration used for most of the development of DUCC consisted of four physical 32-GB machines running 52 simulated nodes of varying memory sizes from 32 to 128-GB each.

To simulate job loads, a simple UIMA-AS job that sleeps for some easily configured length of time was constructed. Another control file is used to generate job specifications requesting randomly-chosen job parameters such as memory requirements, service dependencies, scheduling classes, and so on.

The test suite contains a simple UIMA Analysis Engine called `FixedSleepAE`, and a simple Collection Reader called `FixedSleepCR`. The CR reads a set of sleep times, creates CASs, and ships them to the AEs via DUCC's Job Driver. The CAS contains the time to sleep and various parameters regarding error injection.

The AE receives a CAS, performs error injection if requested, and sleeps the indicated period of time, simulating actual computation but requiring very few physical resources. Hence, many of these may be run simultaneously on relatively modest hardware.

Developers may construct arbitrary jobs by creating a file with sleep times designed to exercise what ever is necessary. DUCC ships with the three primary job collections (test suites) used during initial development. The suites are based on actual workloads and have shown to be very robust for proving the correctness of the DUCC code under stress.

The cluster simulator has been also been run on a 4GB iMac with 8 simulated Agents, an 8GB MacBook with the same configuration, a 32GB iMac with up to 40 simulated Agents. It has also been scaled up to run on 8 45GB Intel nodes running Linux, simulating 20TB of memory.

The rest of this chapter describes the mechanics of using these tools.

15.1 Cluster Simulation

15.1.1 Overview

Cluster-based tools such as DUCC are very hard to test and debug because all interesting problems occur only when the system is under stress. Acquisition of a cluster of sufficient size to expose the interesting problems is usually not practical.

DUCC's design divorces all the DUCC processes from specific IP addresses or node names. ActiveMQ is used as a nameserver and packet router so that all messages can be delivered by name, irrespective of the physical hardware

the destination process may reside upon.

A DUCC system is comprised of three types of processes (daemons):

1. The DUCC management daemons:
   - The Orchestrator (OR). This is the primary point of entry to the system and is responsible for managing the life cycle of all work in the system.
   - The Process Manager (PM). This is responsible for managing message flow to and from the DUCC Agents.
   - The Resource Manager (RM). This is responsible for apportioning system resources among submitted work (jobs, reservations, services).
   - The Service Manager (SM). This is responsible for keeping services active and available as needed.
   - The Web Server (WS). This process listens to all the state messages in the system to provide a coherent view of DUCC to the outside world.
2. The DUCC Node Agents, or simply, Agents. There is one Agent running on every physical node.
3. The ActiveMQ Broker. All message flow in the system is directed through the ActiveMQ broker, with the exception of the CLI, (which uses HTTP).

Normally, the DUCC Agents report the name, IP address, and physical memory of the node they actually do reside upon. This is simply for convenience. It is possible to parametrize the DUCC Agents to report any arbitrary name and address to the DUCC. DUCC components that need to know about Node Agents establish subscriptions to the Agent publications with ActiveMQ and build up their internal structures from the node identities in the Agent publications. Processes which normally establish agent listeners are are the RM, PM, and WS.

It is also possible to parametrize a DUCC agent to cause it to report any arbitrary memory size. Thus, an agent running on a 2GB machine can be started so that it reports 32GB of memory. This parametrization is specifically for testing, of course.

The ability to parametrize agent identities and memory sizes is what enables cluster simulation. A control file is used by start-up scripting to spawn multiple agents per node, each with unique identities.

15.1.2 Node Configuration

A Java properties file is used to configure simulated nodes. There are three types o entries in this file:

nodes This single entry provides the blank-delimited names of the physical nodes participating in the simulated cluster.

memory This single line consists of a blank-delimited set of numbers. Each number corresponds to some memory size, in GB, to be simulated.

node descriptions There are one or more of these. The format of each line is

```
[nodename].[memory] = [count]
```

where

nodename is the name of one of the nodes in the *nodes* line mentioned above.

memory is one of the memory sizes given in the *memory* line mentioned above.

count is the number of simulated agents in the indicated node, with the indicated memory, to be simulated.

For example, the following simulated cluster configuration defines twenty (20) simulated nodes, all to be run on the single physical machine called *agentn*. The simulated nodes contain a mix of 31GB, 47GB, and 79GB memory sizes. There are 7 31GB nodes, 7 47GB nodes, and 6 79GB nodes.

```
# names of nodes in the test cluster
nodes       = agentn
```

```
# set of memory sizes to configure
memory      = 31 47 79

# how to configure memories: node.memsize = count
agentn.31 = 7
agentn.47 = 7
agentn.79 = 6
```

The nodenames generated by this means are the name of the physical node where the agent is spawned, and a numeric id appended, for example,

```
agentn-1
agentn-2
agentn-3
  etc.
```

15.1.3 Setting up Test Mode

During simulation and testing it is desirable and usually required that DUCC run in unprivileged mode, with all processes belonging to a single userid. Unfortunately, this does not exercise any of the multi-user code paths, especially in the Resource Manager.

To accommodate this, DUCC can be configured to run in "test mode", such that work is submitted under "simulated" userid which DUCC treats as discrete IDs. All actual work is executed under the ownership of the tester however.

To establish test mode:

1. Ensure that *ducc.properties* is configured to point to a non-privileged version of *ducc_ling*. Specifically, configure this line in *ducc.properties*

   ```
   ducc.agent.launcher.ducc_spawn_path=/home/ducctest/duccling.dir/amd64/ducc_ling
   ```

   in this example a version of *ducc_ling* known to not have elevated privileges it configured.

2. Configure test mode in *ducc.properties*:

   ```
   ducc.runmode=Test
   ```

   IMPORTANT: Do not start DUCC with *ducc.runmode=Test* if *ducc_ling* has elevated privileges. Test mode bypasses the authentication and authorization checks that are normally used and the system would run completely open.

In test mode, jobs may specify what simulated userid is to be used. Most of DUCC does not pay any attention to the user so this works fine, and the parts that do care about the user are bypassed when *ducc.runmode=Test* is configured.

15.1.4 Starting a Simulated Cluster

DUCC provides a start-up script in the directory `$DUCC_HOME/examples/systemtest` called `start_sim`.

WARNING: Cluster simulation is intended for DUCC testing, including error injection. It is similar to flying a high-performance fighter jet. It is intentionally twitchy. Very little checking is done and processes may be started multiple time regardless of whether is sane to do this.

To start a simulated cluster, use the *start_sim* script:

Description: The *start_sim* script is used to start a simulated cluster.

Usage: *start_sim* options

Options:

-n, –nodelist [nodelist] where the nodelist is a cluster description as described above.

-c –components [component list] . The component list is a blank-delimited list of components including *or, rm, sm, pm, ws, broker* to start an individual component, or *all* to start all of the components. NOTE: It is usually an error to start any of these components more than once. However *start_sim* allows it, to permit error injection.

–nothreading If specified, the command does not run in multi-threaded mode even if it is supported on the local platform.

15.1.5 Stopping a Simulated Cluster

There are two mechanisms for stopping a simulated cluster:

1. *check_ducc -k* This looks for all DUCC processes on the nodes in $DUCC_HOME/resources/ducc.nodes and issues *kill -9* to each process. It then removes the Orchestrator lock file. This is the most violent and surest way to stop a simulated DUCC cluster. In order for this to work, be sure to include the names of all physical nodes used in the simulated cluster in the DUCC configuration file *$DUCC_HOME/resources/ducc.nodes*. It is described in the administration section of the book.
2. *stop_sim* With no arguments, this attempts to stop all the simulated agents and the management daemons using *kill -INT*. It is possible to stop individual agents or management nodes by specifying their component IDs. The kill signals *-KILL, -STOP* and *-CONT* are all supported. This allows error injection as well as a more orderly shutdown than *check_ducc -k*.

Note that *check_ducc* is found in *$DUCC_HOME/admin*. The *stop_sim* script is found in *duccruntime/examples/systemtest*.

The *start_sim* script creates a file called *sim.pids* containing the physical node name, Unix process ID (PID), and component ID (ws, sm, or, pm, rm) of each started DUCC component. In the case of agents, each agent is assigned a number as a unique id. These ids are used with *stop_sim* to affect specific processes. If the cluster is stopped without using *stop_sim*, or if it simply crashes, this PID file will get out of date. Fly more carefully next time!

stop_sim works as follows:

Description The *stop_sim* script is used to stop some or all of a simulated cluster.

Usage: *stop_sim* [options]

Options:

-c, –component [component name] where the name is one of *rm, sm, pm, or. ws,. Kill -INT* is used to enable orderly shutdown unless overridden with -k, -p, or -r as described below.

-i, –instance [instance-id] where the instance-id is one of the agent ids in "sim.pids". *Kill -INT* is used to enable orderly shutdown unless overridden with -k, -p, or -r as described below.

-k, –kill Use *kill -9* to kill the process.

-p, –pause Signal the process with *SIGSTOP*.

-r, –resume Signal the process with *SIGCONT*.

–nothreading If specified, the command does not run in multi-threaded mode even if it is supported on the local platform.

15.2 Job Simulation

15.2.1 Overview

"Real" jobs are highly memory and CPU intensive. For testing and simulation purposes, the jobs need not use anywhere close to their declared memory, and need not consume any CPU at all. The FixedSleepAE is a UIMA analytic that is given a time, in milliseconds, and all it does is sleep for that period of time and then exit. By running many of these in a simulated cluster it is possible to get all the DUCC administrative processes to behave as if there is a real load on the system when in fact all the nodes and jobs are taking minimal resources.

The FixedSleepAE is delivered CASs by the FixedSleepCR. This CR reads a standard Java properties file, using the property "elapsed" to derive the set of sleep times. On each call to the CR's "getNext()" method, the next integer from "elapsed" is fetched, packaged into a CAS, and shipped to ActiveMQ where it is picked up by the next available FixedSleepAE.

The test driver is given a control file with the names of all the jobs to be submitted in the current run, and the elapsed time to wait between submission of each job. Each job name corresponds to a file that is not an actual DUCC specification, but rather the description of a DUCC specification. Each description is a simple Java properties file.

To submit a job, the test driver reads the next job description file derive the number of threads, the simulated user, the desired (simulated) memory for the job, (possibly) the service ID, and the scheduling class for the job. From these it constructs a DUCC job specification and submits it to DUCC.

Scripting is used to read the job meta-descriptors and generate a control file that submits the job set with a large set of variations. The same scripting reads each meta-descriptor and modifies it according to the specific parameters of the run, adjust things such as scheduling class, memory size, etc.

15.2.2 Job meta-descriptors

For each simulated job in a run, a meta-descriptor must be constructed. These may be constructed "by hand", or via local scripting, for example from log analysis. (The packaged meta-descriptors are generated from logs of actual workloads.)

A meta-descriptor must contain the following properties:

tod This specifies a virtual "time of day of submission", starting from time 0, specified in units of milliseconds, when the job is to be submitted. During job generation, this may be used to enforce precise timing of submission of the jobs.

elapsed This is a blank-delimited set of numbers. Each number represents the elapsed time, in milliseconds, for a single work item. There must be one time for each work item. These numbers are placed into CASs by the job's Job Driver and delivered to each Job Process. For example, if this job is to consist of 5 work items of 1, 2, 3, 4 and 5 seconds each, specify

```
elapsed = 1000 2000 3000 4000 5000
```

threads This is the number of threads per Job Process. It is translated to the *process_thread_count* parameter in the job specification.

user This is the name of the user who owns the job. It may be any string at all. If DUCC is started in *test* mode, this will be shown as the owner of the job in the webserver and the logs.

memory This is the amount of memory to be requested for the job, translating to the job specification's *process_memory_size* parameter.

class This is the scheduling class for the job.

machines This is the maximum number of processes to be allocated for the job, corresponding to the *process_deployments_max* parameter.

For example:

```
tod = 0
elapsed = 253677 344843 349342 392883 276264 560153 162850 744822 431210 91188 840262 843378
threads = 4
user = Rodrigo
memory = 20
class = normal
machines = 11
```

All the job meta-descriptors for a run must be placed into a single directory.

15.2.3 *Prepare* Descriptors

A *prepare descriptor* is also a standard Java properties file. This defines where the set of meta descriptors resides, where to place the modified meta-files, how to assign scheduling classes to the jobs, how to apportion memory sizes, how to apportion services, how long the total run should last, and how to compress sleep times.

All parts of the run are randomized, but the randomization can be made deterministic between runs by specifying a seed to the random number generator.

Properties include

random.seed This is the random-number generator seed to be used for creating the run.

src.dir This is the directory containing the input-set of meta-specification files.

dest.dir This is the directory that will contain the updated meta-specification files.

scheduling.classes This is a blank-delimited list of the scheduling classes to be randomly assigned to the jobs.

scheduling.classes.[name] Here, *name* is the name of one of the scheduling classes listed above. The value is a weight, to be used to affect the distribution of scheduling classes among the jobs.

job.memory This is a blank-delimited list of memory sizes to be randomly assigned to each job.

job.memory.[men]] Here, *mem* is one of the memory sizes specified above. The value is a weight, used to affect the distribution of memory sizes among the jobs.

job.services This is a blank-delimited list of a service id, where the id is one of the services specified in the *services.boot* control file.

job.services.[id] Here *id* is one of the ids specified in the job.services line above. The value is a weight, used to affect the distribution of services among the jobs.

submission.spread This is the time, in seconds the set of job submissions is to be spread across. The jobs are submitted at random times such that the total time between submitting the first job and the last job is approximately this number.

compression For each sleep time in the job, divide the actual value by this number. This allows testers to use the actual elapsed time from real jobs, and compress the total run time so it fits approximately into the submission spread.

For example, if a collection of jobs was originally run over 24 hours, but you want to run a simulation with approximately the same type of submission that last only 15 minutes, specify a submission spread of 900 (15 minutes) and a compression of 96.

Here is a sample run configuration file:

```
# control file to create a random-like submission of jobs for batch submission
# This represents jobs submitted over approximately 36 hours real time
# Compression of 96 and spread 920 gives a good 15-20 minute test on test system with
# 136 15GB shares

random.seed                   = 0         # a number, for determinate randoms
```

```
                                            # or TOD, and the seed will use
                                            # current time of day

src.dir                       = jobs.in   # where the jobs are
dest.dir                      = jobs      # where to put prepared jobs

scheduling.classes            = normal    # classes
scheduling.classes.normal     = 100

job.memory                    = 28 37     # memorys to assign
job.memory.28                 = 50
job.memory.37                 = 50

job.services                  = 0 1 2 3 4 5 6 7
job.services.0                = 25
job.services.1                = 25
job.services.2                = 25
job.services.3                = 25
job.services.4                = 25
job.services.5                = 25
job.services.6                = 25
job.services.7                = 25

submission.spread             = 920       # number of *seconds* to try to spread submission over

compression                   = 96        # comporession for timings
```

15.2.4 Services

It is possible to run the FixedSleepAE as a UIMA-AS service, with each job specifying a dependency on the service, and the indicated service doing the actual sleeping on behalf of the job.

These variants on services are supported:

1. Registered services, started by reference,
2. Registered services, started by the simulator,

To use these simulated services, configure a "service boot" file and reference the services from the job generation config file.

Properties required in the service boot file include:

register This specifies registered services. The value is a blank delimited list of pseudo IDs for the registered services.

start This specifies which of the registered services to automatically start. The value is some subset of the pseudo IDS specified under *register*

instances_[id] Here *id* is one of the IDs specified for *submit, register,* or *standalone*. The value is the number of instances of that specific service to set up.

Service pseudo IDs DUCC is packaged with 10 pre-configured services that use the FixedSleepAE. All of these services behave identically, the only difference is their endpoints, which allows the simulated runs to activate and use multiple independent services. Because the endpoints are in the various UIMA XML service descriptors, it is necessary to use exactly these IDs when generating a test run. Thus, the only valid pseudo-ids for service configuration are *0, 1, 2, 3, 4, 5, 6, 7, 8, 9*.

These *service ids* are used on the job configuration file to establish a weighted distribution of service use among the jobs.

Here is a sample service configuration file:

```
# register these services, 2 instances each
register 0 1 2 3
instances_0 2
instances_1 2
instances_2 2
instances_3 2

# start these registered services
start 2 3
```

15.2.5 Generating a Job Set

The *prepare* script, found in $DUCC_HOME/examples/systemtest is used to generate a test run from the control files described above. To use it, execute

```
prepare [config-file]
```

where *config file* is the run description file described above.

This script reads the meta-specification in the *jobs.in* directive of the config-file, generates a set of meta-specification files into the *jobs.out* directory, and creates a control file, *job.ctl*. The *job.ctl* file is used by the simulation driver to submit all the jobs.

15.2.6 Running the Test Driver

A test run is driven from the script *runducc* which resides in the directory *$DUCC_HOME/examples/systemtest*. This script supports a large number of options intended to inject errors and otherwise perturb a run.

To use the test driver, first create a job collection as described above. This will generate a file called *job.ctl* in the test directory containing the *prepare* file.

Then execute:

```
runducc -d jobdir -b batchfile options...
```

where the various parameters and options include:

-d jobdir The jobdir is the directory containing the *prepare* file and the *job.ctl* file as describe in the previous section.

-b batchfile The batchfile is usually *job.ctl* as generated by the prepare script. (This file may be hand-edited to create custom runs outside of the *prepare* script.)

–AE This specifies to run all jobs as CR and AE. This is the default and need not be specified.

–DD This specifies to run all jobs as CR and DD. The jobs are generated as DD-style jobs, as opposed to AE.

–SE cfg This specifies to run all jobs using services, as generated by the *prepare* script. The parameter is the service config file as described above. When specified, the driver starts the services as configured, pauses a bit to let them start up, and generated every job with a dependency on one of the services.

-i time-in-sec If specified, this forces each AE to spend a minimum of the indicated time in it's initialization method (also a sleep). If not specified, the default is 10 seconds. The actual time is controlled by the *-r* (range) option.

–init_fail_cap count This sets the job property *process_initialization_failures_cap* to the the indicated value, to control the number of initialization failures to be tolerated before terminating the job.

–int_timeout seconds This sets the job property *process_initialization_time_max* to the indicated value, to control the time allowed in initialization before failure is reported.

-r time-in-sec This specifies the top range for initialization. The service will spend the time specified in *-i*, PLUS a random value from 1 to the time specified in *-r* in its initialization phase.

–IB The Job Process will leak memory in it's initialization phase until it is killed, hopefully by DUCC, but possibly by the operating system. *Use with care..*

–PB The job Process will leak memory in it's processing phase until it is killed, hopefully by DUCC, but possibly by the operation system. *Use with care.*

-m size-in-gb Memory override. Use this value for all jobs, overriding the value in the generated meta-specification file.

-n max-Number-of-processes Max machine override. If specified, this overrides the configured process max from the job control file. Specify the max as 0 and no maximum will be submitted with the job, causing the scheduler to try to allocated the largest possible number of processes to the job.

-p time-in-seconds If specified the job property *process_per_item_time_max*, which sets a timeout on work items, is set to the indicated time.

-w, –watch Submit every job with the *wait_for_completion* flag. This runs the driver in multi-threaded mode, with each thread monitoring the progress of a job.

-x rate This specifies an expected error rate for execution phase in a job process, from 0-100 (a percentage). When specified, each job process uses a random number generator to determine the probability that is would crash, if if that probability is within the specified rate, it generates a random exception.

-y rate This specifies an expected error rate for initialization phase in a job process, from 0-100 (a percentage). When specified, each job process uses a random number generator to determine the probability that is would crash, if if that probability is within the specified rate, it generates a random exception.

For an expected error-free run, only the -b and -d options are needed.

15.3 Pre-Packaged Tests

Three test suites are provided using the mechanisms described in the previous section:

- A 15-minute run comprising approximately 30 jobs. This includes configuration for single-class submission, mixed class submission, and one configured to maximize resource fragmentation.
- A 30-minute run comprising approximately 33 jobs. This includes a single configuration.
- a 24-hour run comprising approximately 260 jobs. This also includes configurations for single-class submission, mixed classes, and fragmentation. *Note: this run has been reconfigured to run in 12 hours, and has been successfully been configured to complete in 6 hours. This can create a significant load on the DUCC processes.*

The configurations are found in the $DUCC_HOME/examples/systemtest directory and are in sub directories called,

- mega-15-min
- mega-30-min
- mega-24-hour

To run these tests:

1. Create a node configuration. A sample configuration to generate 52 simulated nodes, and which assumes the physical machines for the simulation are called *sys290, sys291, sys292, sys293* and *sys534* is supplied in $DUCC_HOME/examples/systemtest. Change the node names to the names of real machines, making any other adjustments needed.

2. Update your *$DUCC_HOME/resources/ducc.nodes* so that all the real node names specified int the simulated node file are included.
3. Update your *$DUCC_HOME/resources/ducc.properties* so the *ducc.head* is specified as the *real, physical* machine where you will start the simulated cluster.
4. Be sure the *job driver* nodepool, if configured in *$DUCC_HOME/resources/ducc.classes*, specifies the name of one of the simulated nodes. When first running these tests it is usually best that the job driver NOT be configured on a specific node in *ducc.classes* as it can be confusing to get this right on simulated clusters.

   Specifically, in *ducc.classes*, configure the *JobDriver* class thus:

   ```
   Class JobDriver fixed-base { }
   ```

   This allows DUCC to schedule the job driver on any node in the simulated cluster.
5. Generate the job set. For example, to generate the job set for the 15-minute run,

   ```
   cd $DUCC_HOME/examples/systemtest
   ./prepare mega-15-min/jobs.prepare
   ```
6. Start the simulated cluster (Assuming your simulated node file is called *52.simulated.nodes*:

   ```
   cd $DUCC_HOME/examples/systemtest
   ./start_sim -c all -n 52.simulated.nodes
   ```
7. Use the webserver (or for advanced users, log files), to ensure everything came up and the job driver node has been assigned.
8. Start the run:

   ```
   cd $DUCC_HOME/examples/systemtest
   ./runducc -d mega-15-min -b job.ctl
   ```

Chapter 16

DUCC Web Server Customization

This chapter describes how to take advantage of DUCC Web Server plug-in capabilities in order to add local modifications.

Prerequisites:

1. Apache UIMA-DUCC source code
2. Ability to install DUCC (e.g. administrator)

Why would you want to do this? Perhaps you have some related information that your DUCC Web Server could display to the user community. There are considerations for both the server and client sides.

The following discussion is related to the downloaded DUCC source code, specifically the project *uima-ducc-web*.

16.1 Server Side

In package *org.apache.uima.ducc.ws* you will find *DuccPlugins.java* which you can modify or extend.

During Web Server boot time, there are hooks to:

1. process each Job, Reservation, and Service restored from history
2. add additional *http* request handlers

During Web Server run time, there are hooks to:

1. process each Orchestrator publication

16.2 Client Side

In folder */src/main/webapp/root/js/* you will find *ducc.local.js* which you can modify.

During browser page load, there are hooks to:

1. perform additional initialization based on page type

During browser page update, there are hooks to:

1. perform additional update based on page type

16.3 Build and Install

Build a new *uima-ducc-web-[version].jar* comprising the revised *DuccPlugins.class* and any additional dependent classes. Replace the vanilla *$DUCC_HOME/lib/uima-ducc/uima-ducc-web-[version].jar* with the one containing your modifications.

Copy your new ducc.local.js to the installed Web Server's *$DUCC_HOME/webserver/root/js* directory.

Start (or re-start) your Web Server.

You may have to flush you browser's cache to pick up the new *ducc.local.js*.

Chapter 17

Understanding the DUCC logs

17.1 Overview

This chapter provides an overview of the DUCC process logs and how to interpret the entries therein.

Each of the DUCC "head node" processes writes a detailed log of its operation to the directory $DUCC_HOME/logs. The logs are managed by Apache log4j. All logs are managed by a single log4j configuration file

```
$DUCC_HOME/resources/log4j.xml
```

The DUCC logger is configured to check for updates to the log4j.xml configuration file and automatically update without the need to restart any of the DUCC processes. The update may take up to 60 seconds to take effect.

The DUCC logger is loaded and configured through the log4j API such that other log4j configuration files that might be in the classpath are ignored. This also means that log4j configuration files in the user's classpath will not interfere with DUCC's logger.

The logs are set to roll after reaching a given size and the number of generations is limited to prevent overrunning disk space. In general the log level is set to provide sufficient diagnostic output to resolve most issues.

Each DUCC component writes its own log as defined in the following table:

Component	Log Name
Resource Manager	rm.log
Service Manager	sm.log
Orchestrator	or.log
Process Manager	pm.log
Web Server	ws.log
Agent	*[hostname].agent.log*

Because there may be many agents, the agent log is prefixed with the name of the host for each running agent.

The log4j file may be customized for each installation to change the format or content of the log files, according to the rules defined by log4j itself.

The general format of a log message is as follows:

```
Timestamp  LOGLEVEL  COMPONENT.sourceFileName method-name J[Jobid-or-NA] T[TID] text
```

where

Timestamp is the time of the occurrence. By default, the timestamp uses millisecond granularity.

LOGLEVEL This is one of the log4j debug levels, INFO, ERROR, DEBUG, etc.

COMPONENT This identifies the DUCC component emitting the message. The components include

SM Service Manager

RM Resource Manager

PM Process Manager

OR Orchestrator

WS Web Server

Agent Agent

JD Job Driver. These logs are written to the log directory specified in each job submission.

JobProcessComponent Job process, also known as JP. These logs are written to the log directory specifid in each job submission.

sourceFileName This is the name of the Java source file from which the message is emitted.

method-name This is the name of the method in *sourceFileName* which emitted the message.

J[Workid-or-NA] This is the DUCC assigned id of the work being processed, when relevant. If the message is not associated with work, this field shows “N/A”. Some logs (such as JP and JD logs) pertain ONLY to a specific job and do not contain this field.

T[TID] This is the ID of the thread emitting the message. Some logs (such as RM) do not use this field so it is omitted.

text This is the component-specific text content of the message. Key messages are described in detail in subsequent sections.

17.2 Resource Manager Log (rm.log)

The RM log is designed to show all phases of resource scheduling. Much of the flow of a job can be observed in this log alone. The following specific information is available and is explained in more detail below:

- Bootstrap configuration
- Node arrival and missed heartbeats
- Node occupancy
- Job arrival and status updates
- Calculation of job caps
- How-much - fair share
- What-of - host assignment and preemption
- Defragmentation
- Internal schedule
- Published schedule

Most useful messages are emitted under log level INFO but a wealth of details can be seen by incresing the log level to DEBUG or TRACE. To do so, edit the file *$DUCC_HOME/resources/log4j.xml* and change the *priority* value to *debug* (or *trace*) in the stanza similar to that shown here. Within about a minute the logger will pick up the change and increase its log level.

```
<category name="org.apache.uima.ducc.sm" additivity="true">
  <priority value="debug"/>
  <appender-ref ref="smlog" />
</category>
```

17.2.1 Bootstrap Configuration

The RM summarizes its entire configuration when it starts up and prints it to the log to provide context for subsequent data and as verification that the RM is configured in the way it was thought to be. All the following are found in the bootstrap section and are mostly self-explanatory:

- A pretty-print of the class configuration. This is the same as produced by the *check_ducc -c -v* command.
- A summary of all classes, one per line. This is a more concise display and is similar to the DUCC Classes page in the web server.
- A listing of all RM configuration parameters and the environment including things such as the version of Java, the operating system, etc.
- Nodepool occupancy. As host names are parsed from the *ducc.nodes* files, the RM log shows exactly which nodepool each node is added to.

The RM logs can wrap quickly under high load in which case this information is lost.

The following represent key RM logs lines to search for if it is desired to examine or verify its initialization. (Part of the leaders on these messages are removed here to shorten the lines for publication.)

Initial RM start The first logged line of any RM start will contain the string *Starting component: resourceManager*:

```
RM.ResourceManagerComponent- N/A boot  ... Starting Component:  resourceManager
```

RM Node and Class Configuration The first configuration lines show the reading and validation of the node and class configuration. Look for the string *printNodepool* to find these lines:

```
RM.Config- N/A printNodepool   Nodepool --default--
RM.Config- N/A printNodepool      Search Order: 100
RM.Config- N/A printNodepool      Node File: None
RM.Config- N/A printNodepool                 <None>
RM.Config- N/A printNodepool      Classes: background low normal high normal-all nightly-test reserve
RM.Config- N/A printNodepool      Subpools: jobdriver power intel
        ...
```

RM Scheduling Configuration Next the RM reads configures its scheduling parameters and emits the information. It also emits information about its environment: the ActiveMQ broker, JVM information, OS information, DUCC version, etc. To fine this search for the string *init Scheduler*.

```
 init  Scheduler running with share quantum      :  15  GB
 init                         reserved DRAM      :  0  GB
 init                         DRAM override      :  0  GB
 init                         scheduler          :  org.apache.uima.ducc.rm.scheduler.NodepoolScheduler
           ... (more lines) ...

 init                         DUCC home          :  /home/challngr/ducc_runtime
 init                         ActiveMQ URL       :  tcp://bluej537:61617?jms.useCompression=true
 init                         JVM                :  Oracle Corporation 1.7.0_45
 init                         JAVA_HOME          :  /users1/challngr/jdk1.7.0_45/jre
 init                         JVM Path           :  /users/challngr/jdk1.7.0_45/bin/java
 init                         JMX URL            :  service:jmx:rmi:///jndi/rmi://bluej537:2099/jmxrmi
 init                         OS Architecture    :  amd64
 init                         OS Name            :  Linux
 init                         DUCC Version       :  2.0.0-beta
 init                         RM Version         :  2.0.0
```

RM Begins to Schedule The next lines will show the nodes checking in and which nodepools they are assigned to. When the scheduler is ready to accept Orchestrator requests you will see assignment of the JobDriver reservation. At this point RM is fully operational. The confirmation of JobDriver assignment is similar to this:

```
Reserved:
         ID    JobName    User      Class Shares Order QShares NTh Memory nQuest Ques Rem InitWait Max P/Ns
R______7434 Job_Drive   System  JobDriver      1     1       1   0      2      0        0        0
```

17.2.2 Node Arrival and Missed Heartbeats

Node Arrival

As each node "checks in" with the RM a line is printed with details about the node. Some fields are redundant but are produced by different components processing the node arrival and thus serve as confirmation that all parts are operating correctly.

A node arrival entry is of the form:

```
LOGHEADER Nodepool: power Host added: power :  bluej290-18   shares  3 total    9:   48128 <none>
```

where the fields mean (if the field isn't described here, the value is not relevant to node arrival):

LOGHEADER is the log entry header as described above.

Nodepool:power The node is added to the "power" nodepool

bluej290-18 This is the name of the node

shares 3 The number of full shares supported on this machine.

total 9 This is the total shares in the system after this node arrives.

48128 This is the memory, in KB, on that host.

Missed Heartbeats

The DUCC Agents send out regular "heartbeat" messages with current node statistics. These messages are used by RM to determine if a node has failed. If a heartbeat does not arrive at the specified time this is noted in the log as a *missing heartbeat*. If a specific (configurable) number of consecutive heartbeats is missed, the RM marks the node offline and instructs the DUCC Orchestrator to purge the shares so they can be rescheduled.

A missed heartbeat log entry is of the form

```
[LOGHEADER] "*** Missed heartbeat ***" NODENAME count[NN]
```

where the fields mean:

LOGHEADER is the log entry header as described above.

***** Missed heartbeat ***** Indicates this is a missing heartbeat message.

NODENAME This is the name of the (possibly) errant host.

count[N] This is the number of CONSECUTIVE missing heartbeats.

Note that it is not unusual to miss the occasional heartbeat or two due to general network or system load. As soon as a heartbeat is received the count is reset to 0.

If the number of missing heartbeats exceeds the value *ducc.rm.node.stability* configured in *ducc.properties* the node is marked offline and this message is emitted:

```
HEADER "*** ! Notification of node death:" NODENAME
```

If the node recovers and rejoins, the NodeArrives message as described above is emitted.

17.2.3 Node Occupancy

Node occupancy describes, for each node, the capacity of the node, the work assigned to that node, and the number of open shares on that node. The RM writes the node occupancy to its log before assignment of every new schedule. The occupancy can be found under the log header line:

```
[LOGHEADER] Machine occupancy before schedule
```

NOTE: The current node occupancy can be queried interactively with the rm_occupancy command:

```
DUCC_HOME/admin/rm_qoccupancy
```

Sample node occupancy as displayed in the log follows. The header is included in the log.

```
           Name Order Active Shares Unused Shares Memory (MB) Jobs
--------------- ----- ------------- ------------- ----------- ------ ...
 f1n2.bluej.net    16            16             0      255459 206710 206715 207878 206719 207900
 f1n4.bluej.net    16             0            16      255459 <none>[16]
 f7n2.bluej.net    16             0            16      255459 <none>[16]
f9n10.bluej.net    16             0            16      255459 <none>[16]
 f6n1.bluej.net    16             0            16      255459 <none>[16]
 f7n1.bluej.net    16             3            13      255459 203408 [13]
 f7n3.bluej.net    16            16             0      255459 206716 207904 206720 206717 206718
f4n10.bluej.net    16            15             1      255459 209155 208975 209153 209155 [1]
 f7n5.bluej.net    16            16             0      255459 208960
 f1n3.bluej.net    16            16             0      255459 205608 206695 207906 206693 206693
 f1n1.bluej.net    16             3            13      255459 208913 [13]
f6n10.bluej.net    16             3            13      255459 208977 [13]
 f6n7.bluej.net    16             0            16      255459 <none> [16]
 f7n6.bluej.net    16            15             1      255459 209155 209151 206701 209155 [1]
```

The meaning of each column is:

Name The host name.

Order This is the share order of the node. The number represents the number of quantum shares that can be scheduled on this node. (Recall that an actual process may and usually does occupy multiple quantum shares.)

Active Shares This is the number of quantum shares on the node which are scheduled for work.

Unused Shares This is the number of quantum shares available for new work.

Memory This is the real memory capacity of the node, as reported by the node's Agent process.

Jobs Each entry here is the DUCC-assigned id of a job with process assigned to this node. Each entry corresponds to one process. If an ID appears more than once the job has more than one process assigned to the node; see for example, the node **f1n3.bluej.net** with multiple entries for job *206693*.

When no work is assigned to the node, the string <**none**> is displayed.

When there is a number in brackets, e.g. [**13**] for node **f7n1.bluej.net**, the number represents the number of quantum shares available to be scheduled on the node.

17.2.4 Job Arrival and Status Updates

Orchestrator State Arrival On a regular basis the Orchestrator publishes the full state of work which may require resources. This is the prime input to the RM's scheduler and must arrive on a regular basis. Every arrival of an Orchestrator publication is flagged in the log as follows. If these aren't observed every Orchestrator publish interval something is wrong; most likely the Orchestrator or the ActiveMQ broker has a problem.

```
RM.ResourceManagerComponent- N/A onJobManagerStateUpdate  -------> OR state arrives
```

Job State Immediately after the OR state arrival is logged the state of all work needing scheduling is logged. These are always tracked by the *JobManagerConverter* module in the RM and is logged similar to the following. It shows the state of each bit of work of interest, and if that state has changed since the last publication, what that state is.

```
    ...
RM.JobManagerConverter- 7433 eventArrives  Received non-schedulable job, state =  Completed
RM.JobManagerConverter- 7434 eventArrives  [SPR] State:  WaitingForResources -> Assigned
    ...
```

17.2.5 Calculation Of Job Caps

Prior to every schedule, and immediately after receipt of the Orchestrator state, the RM examines every piece of work and calculates the maximum level of resources the job can physically use at the moment. This handles the *expand-by-doubling* function, the *prediction* function, and accounts for the amount of work left relative to the resources the work already possesses.

The curious or intrepid can see the code that implements this in *RmJob.java* method *initJobCap()*.

The calculation is done in two steps:

1. Calculate the projected cap. This uses the prediction logic and amount of work remaining to calculate the *largest* number of resources the job can use, if the system had unlimited resources. This is an upper bound on the actual resources assigned to the job.
2. Adjust the cap down using expand-by-doubling and the initialization state of the work. The result of this step is always a *smaller or equal* number as the projected cap.

The goal of this step is to calculate the largest number of resource the job can actually use at the moment. The FAIR_SHARE calculations may further revise this down, but will never revise it up.

If there is no data yet on the initialization state of work, the rejected cap cannot be calculated and a line such as the following is emitted:

```
RM.RmJob - 7483 getPrjCap  Hilaria Cannot predict cap: init_wait true || time_per_item 0.0
```

If the job has completed initialization the projected cap is calculated based on the average initialization time of all the job processes and the current rate of work-item completion. A line such as this is emitted:

```
RM.RmJob- 7483 Hilaria O 2 T 58626 NTh 28 TI 18626 TR 12469.0 R 2.2456e-03 QR 1868 \
       P 132 F 1736  ST 1433260775524 return 434
```

In this particular line:

7483 is the job id

Hilaria is the job's owner (userid)

O 2 this says this is an *order 2* job: each process will occupy two quantum shares.

T 58626 is the smallest number of milliseconds until a new process for this job can be made runnable, based on the average initialization time for processes in this job, the Orchestrator publish rate, and the *RM prediction fudge.*

Nth This is the number of threads currently executing for this job. It is calculated as the (number of currently allocated processes) * (the number of threads per process).

TI This is the average initialization time in milliseconds for processes in this job.

TR This is the average execution time in milliseconds for work items in this job.

R This is the current rate at which the job is completing work items, calculated as (Nth / TR).

QR The is the number of work items (questions) remaining to be executed.

P This is the projected number of questions that can be completed in the time from "now" until a new process can be started and initialized (in this case 58626 milliseconds from now, see above), with the currently allocated resources, calculated as (T * R).

F This is the number of questions that will remain unanswered at the end of the target (T) period, calculated as (QR - P).

ST This is the time the job was submitted.

return This is the projected cap, the largest number of processes this job can physically use, calculated as (F / threads-per-process).

If the returned projected cap is 0, it is adjusted up to the number of processes currently allocated.

Once the projected cap is calculated a final check is made to avoid several problems:

- Preemption of processes that contain active work but are not using all their threads. This occurs when a job is "winding down" and may have more processes than it technically needs, but all processes still are performing work.
- The job may have declared a maximum number of processes to allocate, which is less than the number it could otherwise be awarded.
- If prediction is being used, revise the estimate down to the smaller of the projected cap and the resources currently allocated.
- If initialization caps are being applied and no process in the job has successfully initialized, revise the estimate down to the initialization cap.
- If expand-by-doubling is being used, potentially revise the estimate down to no more than double the currently allocated processes.

The final cap is emitted in a line such as:

```
RM.RmJob- 7483 initJobCap Hilaria O 2 Base cap: 7 Expected future cap: 434 potential cap 7 actual cap 7
```

In this line:

7483 is the job id.

Hilaria is the job's user name.

O 2 indicates this job uses two quantum shares per processes.

Base cap: This is an upper-bound on the number of processes that can be used in a perfect world. It is calculated by dividing the number of questions by the number of threads per process. It is then revised down by the declared max-processes in the job. In the example above, the job declared max-processes of 7.

Expected future cap This is the projected cap, described above.

Potential cap This is the base cap, possibly revised downward by the future cap, if it is projected that fewer processes are would be useful.

Actual cap This is the assigned maximum processes to be scheduled for this job, possibly adjusted based on the initialization status of the job and the expand-by-doubling policy.

The *actual cap* is the one used to calculate the job's FAIR_SHARE and is always the the largest number of processes usable in a perfect world. Note that the FAIR_SHARE calculation may result in further reduction of this number.

17.2.6 The "how much" calculations

The RM log includes a section that details the fair-share calculations. The details of this are rather involved and out-of-scope for this section. Interested parties are welcome to read the scheduler source, in the file *NodePoolScheduler.java*, methods *countClassShares, countJobShares, countUserShares*, and *apportion_qshares*.

The logs reveal the inputs to each of the methods above. The overall logic is as follows and can be followed in the logs.

- All job classes of equal priority are bundled together and handed to the *countClassShares* method. This method assigns some number of shares to each class based on the weighted fair-share logic, using the configured class weights. The start of this can be seen under log lines similar to this:

  ```
  INFO RM.NodepoolScheduler- N/A apportion_qshares  countClassShares RmCounter Start
  ```

- All users for each class are passed to the *countUserShares* method and then assigned some number of shares from the pool of shares assigned to the class, again using the fair-share computations, but with equal weighs. The start of this can be seen under log lines similar to this:

  ```
  INFO RM.NodepoolScheduler- N/A apportion_qshares  countJobShares RmCounter Start
  ```

- All jobs for each user are passed to the *countJobShares* method and assigned some number of shares from the pool assigned to the user, using the fair-share calculator with equal weights. The start of this can be seen under log lines similar to this:

  ```
  INFO RM.NodepoolScheduler- N/A apportion_qshares  countUserShares RmCounter Start
  ```

- The method *apportion_qshares* is the common fair-share calculator, used by the three routines above.

17.2.7 The "what of" calculations

These calculations are also too involved to discuss in detail for this section.

Interested parties may look in *NodePoolScheduler.java*, method *whatOfFairShare*, and *NodePool.java* method *traverseNodepoolsForExpansion* to see details.

The logs track the general flow through the methods above and generally contain enough information to diagnose problems should they arise.

The key log message here, other than those sketching logic flow, shows the assignment of specific processes to jobs as seen below.

```
RM.NodePool- 7483 connectShare  share bluej290-12.461 order 2 machine \
                  bluej290-12        false     2             0             2       31744 <none>
```

This shows job *7483* being assigned a process on host *bluej290-12* as RM share id *461*, which consists of *2 quantum shares* (order 2). Host bluej290-12 is a 32GB machine with *31744* KB of usable, schedulable memory.

17.2.8 Defragmentation

The RM considers the system's memory pool to be fragmented if the counted resources from the the "how much" phase of scheduling cannot be fully mapped to real physical resources in the "what of" phase. In short, the "how much" phase assumes an ideal, unfragmented virtual cluster. The "what of" phase may be unable to make the necessary physical assignments without excessive preemption of jobs that are legitimately at or below their fair share allocations.

Intuitively, the "how much" phase guarantees that if you could do unlimited shuffling around of the allocated resources, everything would "fit". The system is considered fragmented if such shuffling is actually needed. The defragmentation processes attempts that shuffling, under the constraint of interrupting the smallest amount of productive work possible.

One scheduling goal, however, is to attempt to guarantee every job gets at least some minimal number of it's fairly-counted processes. This minimal number is called the defragmentation threshold. and is configured in ducc.properties. This threshold is used to rigorously define "smallest amount of productive work" as used in the previous paragraph. The defragmentation threshold is used in two ways:

1. Attempt to get every work request resources allocated at least up to the level of the defragmentation threshold.

2. Never steal resources beyond the defragmentation threshold during the “take from the rich” phase of defragmentation, described below.

To accomplish this, a final stage, “defragmentation”, is performed before publishing the new schedule to the Orchestrator for deployment.

Defragmentation consists of several steps. The details are again involved, but an understanding of the logic will make following the log relatively straightforward.

- Examine every job and determine whether it was assigned all the processes from the “how much” phase. If not, it is marked as POTENTIALLY NEEDY.

  This step is logged with the tag *detectFragmentation*.

- Examine every POTENTIALLY NEEDY job to determine if there are sufficient preemptions pending such that the “how much” phase will be able to complete as soon as the preemptions complete. If not, the job is marked ACTUALLY NEEDY.

  This step is also logged with the tag *detectFragmentation*.

- For every job marked ACTUALLY NEEDY, examine all jobs in the system already assigned shares to determine which ones can donate some resources to the ACTUALLY NEEDY jobs. These are typically jobs with more processes than their FAIR SHARE, but which, in a perfect, unfragmented layout, would be allocated more resources. These jobs are called *rich* jobs.

  This step is logged with the tags *insureFullEviction* and *doFinalEvictions*.

- Attempt to match allocations from “rich” jobs with jobs that are ACTUALLY NEEDY. If the ACTUALLY NEEDY job is able to use one of the “rich job” allocations, the allocation is scheduled for preemption. (Note there are many reasons that a rich job may not have appropriate resources to donate: mismatched nodepool, physical host too small, not preemptable, etc.).

  This step is logged with the tag *takeFromTheRich*. If this step has any successes, the log will also show lines with the tags *clearShare* and *shrinkByOne* as the resources are scheduled for reuse.

- The needy job is placed in a list of jobs which are given the highest priority for assignment of new processes, at the start of each subsequent scheduling cycle, until such time as they are no longer needy.

  This step is logged with the tag *Expand needy*.

Those who wish to see the details of defragmentation can find them in *NodepoolScheduler.java*, starting with the method *detectFragmentation* and tracing the flows from there.

17.2.9 Published Schedule

The schedule gets printed to the log twice on every scheduling cycle. The first form is a pretty-printed summary of all known jobs, showing which ones are getting more resources, *expanding*, those which are losing resources, *shrinking*, and those which are not changing, *stable*.

The second form is a *toString()* of the structure sent to the Orchestrator, showing the exact resources currently assigned, added, or lost this cycle.

The pretty-printed schedule This entry is divided into five sections. Each section contains one line for each relevant job, with largely self-explanatory headers. An example follows (wrapped here so it fits within a printed page):

```
       ID JobName  User  Class   Shares Order QShares NTh Memory nQuest Ques Rem InitWait Max P/Nst
J____7485 mega-2   bob   low          7     2      14   4     24  11510    11495    false         7
J____7486 mega-1   mary  normal      93     2     186   4     28  14768    14764    false        93
```

Here,

ID is the unique DUCC ID of the work, prefixed with an indication of what kind of work it is: Job (J), a Service (S), a Reservation (R), or Managed Reservation (M).

JobName is the user-supplied name / description of the job.

User is the owner of the work.

Class is the scheduling class used to schedule the work.

Shares is the number of allocations awarded, which might be processes, or simply reserved space. It is a human-readable convenience, calculated as (Order * QShares).

Order is the number of share quanta per allocation.

QShares is the total quantum shares awarded to the work.

Nth is the declared number of threads per process.

Memory is the amount of memory in GB for each allocation.

nQuest is the number of work items (questions) for the job, where relevant.

Ques Rem is the number of work items not yet completed.

InitWait is either *true* or *false*, indicating whether at least one process has successfully completed initialization.

Max P/Nst is the job-declared maximum processes / instances for the job.

The five subsections of this log section are:

Expanded This is the list of all work that is receiving more resources this cycle.

Shrunken This is the list of work that is losing resources this cycle.

Stable This is the list of work whose assigned resources do not change this cycle.

Dormant This is the list of work that is unable to receive any resources this cycle.

Reserved This is the list of reservations.

The Orchestrator Structure This is a list containing up to four lines per scheduled work.

The specific resources shown here are formatted thus:

```
hostname.RM share id^Initialization time
```

The *hostname* is the name of the host where the resource is assigned. The *RM Share* is the unique (to RM only) id of the share assigned to this resource. The *Initialization time* is the amount of time spent by the process residing within this resource in its initialization phase.

The lines are:

1. The type of work and it's DUCC ID, for example:

   ```
   Reservation 7438
   ```

2. The complete set of all resources currently assigned to the work, for example:

   ```
   Existing[1]: bluej537-7-73.1^0
   ```

   The resources here include all resources the RM tracks as being owned by the job, including older resources, newly assigned resources, and resources scheduled for eviction. The specific resources which are being added or removed are shown in the next lines.

3. The complete set of resources the RM has scheduled for eviction, but which are not yet confirmed freed. For example, we see 7 resources which have been evicted:

```
Removals[7]: bluej290-11.465^19430 bluej290-12.461^11802 bluej290-4.460^12672 \
             bluej290-5.464^23004  bluej290-2.467^22909 bluej290-7.463^20636 \
             bluej290-6.466^19931
```

4. The complete set of resources which are being added to the work in this cycle. For example:

```
Additions[4]: bluej291-43.560^0 bluej291-42.543^0 bluej290-23.544^0 bluej291-44.559^0
```

In most cases, if resources cannot be awarded, this section also shows the reason string which is published for the benefit of the web server and the Orchestrator's job monitor:

```
 Job         7487 Waiting for defragmentation.
Existing[0]:
Additions[0]:
Removals[0]:
```

In some cases, it is possible that a job will show BOTH Additions and Removals. This usually occurs as a result of the defragmentation step. The job will have been found in need of new resources during the initial fair-share computation but later during fragmentation, it is also found to be a "rich" job which must donate resources to under-allocated work. Not all the processes belonging to the "rich" job may be appropriate for the poor job, in which case they will be allowed to expand even as it is donating some to the under-allocated work.

This can also occur if resources were previously preempted, for some reason the preemption is taking a long time. Since then other resources have become freed and the can now re-expand. It is not possible to reverse a preemption (because the actual state of the preemption is not knowable) so both expansion and shrinkage can be in progress for the same job.

17.3 Service Manager Log (sm.log)

The service manager log shows the events involved in managing services. These events include

- Bootstrap configuration
- Receipt and anlysis of Orchestrator state
- CLI requests: register, modify, start, stop, unregister, etc.
- Dispatching / startup of service instances
- Progression of Service state
- Starting and logging of pingers

To enable finer-grained messages, edit the file *$DUCC_HOME/resources/log4j.xml* and change the *priority* value to *debug* (or *trace*) in the stanza similar to that shown here. Within about a minute the logger will pick up the change and increase its log level.

```
<category name="org.apache.uima.ducc.sm" additivity="true">
  <priority value="debug"/>
  <appender-ref ref="smlog" />
</category>
```

17.3.1 Bootstrap configuration

Initial SM start The first logged line of any RM start will contain the string *Starting component: serviceManager*:

```
SM.ServiceManagerComponent - N/A boot  ... Starting Component:  serviceManager
```

This is followed by a summary of the environment in which the Service Manager is running, including configuration data, information about the JRE and about hosting hardware.

```
  Service Manager starting:
      DUCC home              :  /home/challngr/ducc_runtime
      ActiveMQ URL           :  tcp://bluej537:61617?jms.useCompression=true

      JVM                    :  Oracle Corporation 1.7.0_45
      JAVA_HOME              :  /users1/challngr/jdk1.7.0_45/jre
      JVM Path               :  /users/challngr/jdk1.7.0_45/bin/java
      JMX URL                :  service:jmx:rmi:///jndi/rmi://bluej537:2102/jmxrmi

      OS Architecture        :  amd64
      Crypto enabled         :  true

      Test mode enabled      :  true

      Service ping rate      :  15000
      Service ping timeout   :  15000
      Service ping stability :  10
      Default ping class     :  org.apache.uima.ducc.cli.UimaAsPing

      Init Failure Max       :  1
      Instance Failure Max   :  3
      Instance Failure Window :  10

      DUCC Version           :  2.0.0-beta
      SM Version             :  2.0.0
```

Initialize the service registry Following this is are entries showing the reading and internal initialization of the service registry, for example:

```
SM.ServiceHandler - 411 ServiceStateHandler.registerService  \
                    adding UIMA-AS:FixedSleepAE_1:tcp://bluej537:61617 411
```

Wait for the Resource Manager and Orchestrator to become ready The Service Manager waits until the Resource Manager and Orchestrator are ready. This usually results in lines similar to the following, which are normal and expected. If these lines continue for more than a few minutes it is possible that some part of DUCC has not correctly started:

```
SM.ServiceManagerComponent - N/A orchestratorStateArrives \
                             Orchestrator JD node not assigned, ignoring Orchestrator state update.
```

Begin work When the Service Manager is ready for business, lines similar to the following are emitted on a regular basis:

```
INFO SM.ServiceManagerComponent - N/A processIncoming  ===== Orchestrator State Arrives =====
```

17.3.2 Receipt and analysis of Orchestrator State

17.3.3 CLI Requests

CLI Requests are all logged at log level INFO. Initial receipt of the command is tagged with the string *ServiceManagerComponent* and contains the name of the command issued. Information pertinent to the command issued is logged. For example, this shows registration of a new service.

```
INFO SM.ServiceManagerComponent - 428 register  ServiceRegisterEvent \
     [ninstances=2, autostart=Unset, \
     endpoint=UIMA-AS:FixedSleepAE_5:tcp://bluej537:61617, user=challngr]
```

Some commands require service ownership or administrative authority. Here we show a service being started, and the authorization check being made.

```
INFO SM.ServiceManagerComponent - N/A start  Starting service ServiceStartEvent \
     [friendly=430, user=challngr, instances=-1]
INFO SM.ServiceHandler - 430 authorized  start request from challngr allowed.
```

17.3.4 Dispatching / Startup of Service Instances

The full set of startup messages is logged when an instance is dispatched to DUCC to be scheduled and started. These starts are usually preceeded by a message or two indicating what triggered the start. For instance, this shows service 427 being started because it was referenced by job 7676:

```
INFO SM.ServiceSet - 427 reference  Reference start requested by  7676
INFO SM.ServiceSet - 427 reference   References job/service 7676 count[1] implementors [0]
INFO SM.ServiceSet - 427 reference  Reference starting new service instances.
```

This is followed by a line indicating how many instances are to be started:

```
INFO SM.ServiceSet - 427 start  Starting instance. Current count 0 needed 2
```

Shortly thereafter the *stdout* messages from the submission to the DUCC Orchestrator are shown. These are tagged with the keyword *ServiceInstance*.

```
INFO SM.ServiceInstance - 427 start  START INSTANCE
INFO SM.ServiceInstance - 427 start  Start stdout: 050 ducc_ling Version 1.1.2 \
                          compiled Aug  4 2014 at 06:45:31
```

A few lines later the DUCC-assigned ID of the instance is shown, in this case, ID 7677, for service 427.

```
INFO SM.ServiceInstance - 427 start  Start stdout: Service instance 7677 submitted
INFO SM.ServiceInstance - N/A start  Request to start service 427 accepted as service instance  7677
```

THe next lines show the environment used for the service submit.

```
INFO SM.ServiceInstance - 427 start  Start stdout: 1104 Running with user and group: \
     id 2087 gid 2001 eid 2087 egid 2001
INFO SM.ServiceInstance - 427 start  Start stdout: 300 Bypassing redirect of log.
INFO SM.ServiceInstance - 427 start  Start stdout: 4050 Limits:      CORE soft[1024] hard[-1]
INFO SM.ServiceInstance - 427 start  Start stdout: 4050 Limits:       CPU soft[-1] hard[-1]
INFO SM.ServiceInstance - 427 start  Start stdout: 4050 Limits:      DATA soft[-1] hard[-1]
       ... (more environment) ...
INFO SM.ServiceInstance - 427 start  Start stdout: Environ[0] = DUCC_SERVICE_INSTANCE=0
INFO SM.ServiceInstance - 427 start  Start stdout: Environ[1] = JAVA_HOME=/opt/ibm-jdk1.7
INFO SM.ServiceInstance - 427 start  Start stdout: Environ[2] = LESSCLOSE=lessclose.sh %s %s
INFO SM.ServiceInstance - 427 start  Start stdout: Environ[3] = ENV=/etc/bash.bashrc
```

Next the exact command line executed to submit the isntance is shown. If the submission is successful, see *START INSTANCE COMPLETE*. The exact progression of the scheduling and deployment of an instance is not shown in this log. If it is desired to observe that, look in the RM log, searching on the service instance id. In the examples shown here, that id is 7677.

```
INFO SM.ServiceInstance - 427 start  Start stdout: 1000 Command to exec: /opt/ibm-jdk1.7/bin/java
INFO SM.ServiceInstance - 427 start  Start stdout:     arg[1]: -cp
INFO SM.ServiceInstance - 427 start  <INHIBITED CP>
INFO SM.ServiceInstance - 427 start  <INHIBITED CP>
INFO SM.ServiceInstance - 427 start  Start stdout:     arg[4]: --specification
```

```
INFO SM.ServiceInstance - 427 start  Start stdout:     arg[5]: /home/challngr/ducc_runtime/state/services/4
INFO SM.ServiceInstance - 427 start  Start stdout:     arg[6]: --service_id
INFO SM.ServiceInstance - 427 start  Start stdout:     arg[7]: 427
INFO SM.ServiceInstance - 427 start  Start stdout: 1001 Command launching...
INFO SM.ServiceInstance - 427 start  Start stdout: Service instance 7677 submitted
INFO SM.ServiceInstance - N/A start  Request to start service 427 accepted as service instance  7677
INFO SM.ServiceInstance - 427 start  START INSTANCE COMPLETE
```

The state progression of the service as it starts is then available, as described in the next section.

17.3.5 Progression of Service State

The state of each service as it starts and stops is tracked at log level INFO. To find the state progression for any particular service search on *service-id setState* where service-id is the ID of the service.

Here we show the progression for service 427 as it is started and progresses to fully functional (there are other lines logged between these of course). Note that if a service has multiple instances defined, the overall service state is determined by an aggregate of the states of the individual instances. In these messages, the "Inst" field of each message shows DUCC ID of the most recently updated service instance that triggered the overall state change.

```
SM.ServiceSet - 427 setState  State update \
                              from[Stopped] to[Starting] via[Starting] Inst[7677/Received]
SM.ServiceSet - 427 setState  State update \
                              from[Starting] to[Initializing] via[Initializing] Inst[7677/Initializing]
SM.ServiceSet - 427 setState  State update \
                              from[Initializing] to[Waiting] via[Waiting] Inst[7677/Running]
SM.ServiceSet - 427 setState  State update \
                              from[Waiting] to[Available] via[Available] Inst[7677/Running]
```

17.3.6 Starting and Logging Pingers

When a pinger is started (or restarted) the event is logged and tagged with the string *service-id startPingThread*. Following are a few lines stating the name of the pinger and the Java class used to implement the pinger. For example:

```
INFO SM.ServiceSet - 430 startPingThread  Starting service monitor.
INFO SM.PingDriver - 430 find RegisteredPinger  \
     Loading site-registered service monitor from org.apache.uima.ducc.cli.UimaAsPing
INFO SM.PingDriver - 430 <ctr>  Using ping class org.apache.uima.ducc.cli.UimaAsPing
```

If the pingers do not declare loggers, their *stdout* and *stderr* are captured in the logs, under the tag *handleStatistics*. As well, every ping is recorded with its "info" string, making it possible to see the exact state of the pinger. For example:

```
INFO SM.PingDriver - 411 handleStatistics  Ping ok:  UIMA-AS:FixedSleepAE_1:tcp://bluej537:61617 \
                         Alive[true] Healthy[true] + Info: QDEPTH[0] AveNQ[0] Consum[26] Prod[0] \
                         minNQ[0] maxNQ[0] expCnt[0] inFlt[0] DQ[0] NQ[0] NDisp[0] \
                         MetaNode[192.168.4.36] MetaPid[8892:67]
```

Pingers always return state to the Service Manager and some of that state affects the SM's operation; in particular, pingers can start and stop specific service instances or change the autostart setting for a service. They also must return success and failure status to the SM. The ping state is logged under the tag *service-id signalRebalance* as seen below. If *Additions:* or *Deletions:* is non-zero, you can expect to see SM automatically start or stop specific instances for the service.

```
INFO SM.ServiceSet - 430 signalRebalance  PING: Additions: 0 deletions: 0 \
     excessive failures: false implementors 1 references 0
```

17.3.7 Publishing State

By default the SM log does not include the state as published to the Orchestrator because it can be voluminous and cause the logs to become cluttered and to wrap too fast. It may be necessary to increas the log level to DEBUG as described at the start of this section.

The published state will be emitted with lines similar to the following. Most entries will show “Available”, which means either they do not depend on the Service Manager, or the service they depend on is in state Available. If there is some exceptional condition pertaining to a job, that is shown. In the sample below, Job 251214 is waiting for a specific service whose state itself is “waiting”.

```
INFO SM.ServiceHandler - 251261 handleModifiedobs  No service dependencies, no updates made.
INFO SM.ServiceManagerComponent - N/A publish  Publishing State, active job count = 102
INFO SM.ServiceManagerComponent - N/A publish  Service Map
Job 251159 Service state Available
Job 251263 Service state Available
Job 251192 Service state Available
Job 251214 Service state Waiting \
    [UIMA-AS:fastqa-Dedicated-Staging-2015-08_cache:tcp://broker42:62616 : waiting]
```

17.4 (Orchestrator Log or.log)

To be filled in.

17.5 Process Manager Log (pm.log)

To be filled in.

17.6 Agent log Log (hostname.agent.log)

To be filled in.