

DataCutter Version 3.2

User's Guide

Michael Beynon, Umit Catalyurek, Mike Gray,
Christian Hansen, Shannon Hastings, Tahsin Kurc,
Stephen Langella, Evan Powers, Joel Saltz,
Alan Sussman, Mike Welsh

Department of Biomedical Informatics
Ohio State University
Columbus, OH 43210

Department of Computer Science
University of Maryland
College Park, MD 20742

Contents

1	Legal Information	4
1.1	License Agreement	4
2	Introduction	5
2.1	What Can DataCutter Be Used For?	5
2.2	Knowledge Prerequisite	6
3	Installation	7
3.1	Obtaining DataCutter	7
3.2	Building DataCutter	7
3.2.1	Required Software	7
3.2.2	Optional Software	7
3.2.3	Building DataCutter with CMake	8
4	Quick Start	10
4.1	What do I need to know?	10
4.2	Using dcshell to Launch DataCutter	10
4.3	Dynamic Listen Ports	12
4.4	Sample Application	12
4.5	Running GridArrayAverager	13
4.6	Shutting Down DataCutter	13
5	Configuration	14
5.1	Configuration File	15
5.2	Directory Daemon	16
5.3	Communication Layer	16

5.4	Queue Sizes	17
5.5	Stream Policies	17
5.6	Development Debugging	17
5.7	X11 Remote Display	19
5.8	Build Options	20
6	Application Development	21
6.1	Creating a simple application	21
6.2	Include Files	25
6.3	Error Reporting	25
6.4	Compilation	25
7	Filter Development	26
7.1	Filter Creation	26
7.1.1	Buffers and Work	26
7.1.2	Initialization	27
7.1.3	Processing	28
7.1.4	Finalization	32
7.2	Registering Filters	32
8	DataCutter Extension Packages	35
8.1	XML Process Models and Filter Configuration	35
8.1.1	XML for describing filter libraries	35
8.1.2	XML for modeling processes	36
8.1.3	XML for configuring filters	37
8.2	Globus	38
8.2.1	Enabling Globus Extensions	38

8.2.2	User Certificates	39
8.2.3	Using GRAM to Start DataCutter Daemons	39
8.2.4	Secure Communication with GSS	39
8.2.5	MDS	39
8.3	Storage Resource Broker (SRB)	40
8.3.1	SRB Examples	40
8.3.2	About the Files	42
8.4	Network Weather Service (NWS)	43

1 Legal Information

1.1 License Agreement

The following license agreement is applied to any and all files included in this release of the software.

```
/*-----  
* The contents of this file are subject to the Mozilla Public License Version  
* 1.1 (the "License"); you may not use this file except in compliance with  
* the License. You may obtain a copy of the License at  
* http://www.mozilla.org/MPL/  
* Software distributed under the License is distributed on an "AS IS" basis,  
* WITHOUT WARRANTY OF ANY KIND, either express or implied. See the License for  
* the specific language governing rights and limitations under the License.  
*  
* The Original Code is Initial code.  
* The Initial Developer of the Original Code is OSU Biomedical Informatics and  
* UMD Department of Computer Science. Portions created by the Initial  
* Developer are Copyright (C) 2003 the Initial Developer. All Rights Reserved.  
* Contributor(s):  
* Michael Beynon  
* Umit Catalyurek  
* Mike Gray  
* Christian Hansen  
* Shannon Hastings  
* Tahsin Kurc  
* Stephen Langella  
* Evan Powers  
* Joel Saltz  
* Alan Sussman  
* Mike Welsh  
*  
* Alternatively, the contents of this file may be used under the terms of  
* either the GNU General Public License Version 2 or later (the "GPL"), or  
* the GNU Lesser General Public License Version 2.1 or later (the "LGPL"),  
* in which case the provisions of the GPL or the LGPL are applicable  
* instead of those above. If you wish to allow use of your version of this  
* file only under the terms of either the GPL or the LGPL, and not to allow  
* others to use your version of this file under the terms of the NPL, indicate  
* your decision by deleting the provisions above and replace them with the  
* notice and other provisions required by the GPL or the LGPL. If you do not  
* delete the provisions above, a recipient may use your version of this file  
* under the terms of any one of the NPL, the GPL or the LGPL.  
*-----*/
```

2 Introduction

As networks connecting computational resources get faster, an increasing number of applications are making use of collective computational resources across local and wide-area networks. One of the consequences of this ability is that scientific and engineering simulations can now generate unprecedented amounts of data. In addition, vast amounts of data are being gathered by advanced sensors and instruments at geographically distributed locations, such as satellites, microscopes, and medical imagers. At the same time, disks continue to become larger and cheaper as they become commoditized; large disk-based storage systems are increasingly easy and inexpensive to set up.

The availability of low cost systems has greatly enhanced a scientist's ability to generate large scale scientific datasets from, store them at, and share them with many disparate, distributed, and heterogeneous locations and parties. This trend results in very large datasets distributed across a network of storage and computational resources.

The primary goal of generating data through large scale simulations or sensors is to better understand the causes and effects of physical phenomena. Understanding can be achieved by querying, mining and analyzing such massive and complex datasets so that the scientist can gain insights into the problem at hand. This is accomplished in two phases through data exploration and analysis. For the data exploration phase, the data of interest is extracted from all relevant datasets through the use of efficient indexing schemes to quickly locate the data items that satisfy a request. In the data analysis phase, application-specific knowledge is used to process and transform the data into a new data product that can be more efficiently consumed by another program or analyzed interactively.

In this project we have been developing a framework, called DataCutter, that is designed to enable exploration and analysis of scientific datasets in distributed and heterogeneous environments. The programming model in DataCutter, called filter-stream programming, represents components of a data-intensive application as a set of filters. Each filter can potentially be executed on a different host across a wide-area network. Data exchange between any two filters is achieved by streams, which are uni-directional pipes that deliver data in fixed size buffers. DataCutter provides a core set of services, on top of which application developers can implement more application-specific services or combine with existing Grid services such as metadata management, resource management, and authentication services. The main design objective in DataCutter is to extend and apply features of the Active Data Repository (ADR), namely support for accessing subsets of datasets via range queries and user-defined filtering operations, for very large datasets in a shared distributed computing environment.

2.1 What Can DataCutter Be Used For?

DataCutter is a framework that allows applications to describe and implement graphs of components that can execute in a distributed environment. The components operate in a data-flow style, where they repeatedly read buffers from their inputs, perform application-defined processing on the buffer, and then write it to the output stream. The DataCutter filtering service manages the complexity to provide a fairly simple programming model to the application, while still achieving good performance. DataCutter provides support for data exploration and analysis through distributed computation resource management.

This documentation is intended to describe the current implementation (DataCutter3.0) in sufficient detail to enable the reader to configure the DataCutter framework and develop a new application.

2.2 Knowledge Prerequisite

A user of DataCutter should be comfortable with the UNIX terminal and programming in C/C++. The rest of the information necessary to use DataCutter can be learned by carefully reading this document.

3 Installation

DataCutter has been compiled and tested on several platforms. The following table lists these platforms.

Platforms
RedHat Linux 7.x, 8.0, running on x86 processors
Linux kernel 2.4, with GCC version 2.95 or later, running on x86 processors
SunOS 5.8, running on SPARC processors
AIX
IA64

Table 1: Platforms

3.1 Obtaining DataCutter

The installation of DataCutter requires building the source on the system on which DataCutter will be installed. The DataCutter source is contained in a gzipped tar file which may be downloaded from the DataCutter website, <http://www.datacutter.org>.

3.2 Building DataCutter

3.2.1 Required Software

DataCutter uses CMake, a cross platform open-source make system. CMake is used to control the software compilation process, using platform and compiler independent configuration files. CMake generates native makefiles and workspaces that can then be used in DataCutter supported environments to build DataCutter. CMake can be downloaded from the CMake website, <http://www.cmake.org>

3.2.2 Optional Software

DataCutter has been integrated with several external packages, which add extra functionality and support to the basic DataCutter package. Although none of these external packages are required, these extensions provide enhanced security, usability, performance, and extendability to DataCutter. More information on how to use these extensions can be found in the Extensions section of this document. Below is a list of the optional software required by these extensions. There is more information on these extensions in Section 8.

- `xerces-c++ 2.2` - Used for XML parsing, required for using XML configurable filters, the XML generic console and the XML daemon console. Xerces can be downloaded from <http://xml.apache.org>. Configuration instructions are in Section 8.1.

- Globus 2.2.2 - Toolkit containing the fundamental technologies needed to build computational grids. Globus provides DataCutter with an alternative method for starting up processes. More importantly Globus provides DataCutter with a method of secure communication between DataCutter filters. Globus can be downloaded from <http://www.globus.org>. Configuration instructions are in Section 8.2.
- Storage Resource Broker (SRB) 2.0 - Middleware that provides a uniform interface for connecting to heterogeneous data resources over a network. SRB provides DataCutter filters with an alternative method for accessing datasets. Through SRB, datasets can be accessed based on their attributes rather than physical location. SRB can be downloaded from <http://www.npaci.edu/DICE/SRB/>. Configuration instructions are in Section 8.3.
- Network Weather Service (NWS) 2.2.1 - A distributed system that monitors and forecasts the performance of network and computational resources. NWS can be used by DataCutter to help decide which hosts to run filters on so that optimal performance is achieved. NWS can be downloaded from <http://nws.cs.ucsb.edu/>. Configuration instructions are in Section 8.4.

3.2.3 Building DataCutter with CMake

This section will explain how to build DataCutter using CMake. At this point it is assumed that CMake is installed on your system and that the CMake executable is in your path. The first step is to extract the DataCutter source from the assumed DataCutter gzipped tar file(DataCutter3.0.tar.gz):

```
> gzip -d DataCutter3.0.tar.gz
> tar -xvf DataCutter3.0.tar
```

Extracting the DataCutter source will create the directory DataCutter3.0. From this point forward we will refer to the full path of this directory as `DATA CUTTER_ROOT`. For example if you extracted the DataCutter source in the directory `/home/johndoe` then your `DATA CUTTER_ROOT` will be `/home/johndoe/DataCutter3.0`. The `DATA CUTTER_ROOT` is only used within the context of this document; when you are given directions containing `DATA CUTTER_ROOT`, you should replace it with the actual path.

The DataCutter3.0 directory contains the subdirectories `nuevodc` and `dcbuild`, the `nuevodc` directory contains the DataCutter source and documentation. The `dcbuild` directory is initially empty but will contain the binaries after DataCutter is built. Before building DataCutter one must run CMake which will create the makefiles needed by your platform. This can be done as follows:

```
> cd DATA CUTTER_ROOT/dcbuild
> cmake ../nuevodc
```

Once CMake has finished configuring the build, DataCutter may be built as follows:

```
> gmake all
```

After DataCutter is built you will notice that the `DATA CUTTER_ROOT/dcbuild` directory contains several sub-directories. The most important of these subdirectories is the `bin` and `lib` directories. The `lib` directory contains the DataCutter shared object library, which is needed to link against when developing a DataCutter application. The `lib` directory also contains the shared object libraries with the example filters distributed with DataCutter. The `bin` directory contains all the DataCutter executables, it is important that this directory is included in your path.

It is highly recommended that you do NOT build in the source directory!

4 Quick Start

This section gives a brief overview about how to start and execute a sample DataCutter application.

4.1 What do I need to know?

There are three executables that are important to understand before trying to use DataCutter.

1. `dird` is the Directory Daemon. The Directory Daemon is a service that maintains a list of nodes that DataCutter applications may be distributed across. The `dird` may also store additional information that is needed by the DataCutter runtime system or DataCutter application. Before running any applications on DataCutter a `dird` must be started. By default the Directory Daemon listens on port “6100 + user’s UNIX uid”, in order to prevent simultaneous users from colliding with each other.
2. `appd` is the application daemon. This is the service that controls the applications running on each of the nodes of the distributed system. Any node wishing to be utilized by DataCutter must run an `appd`. When an `appd` starts it registers itself with the `dird`. Thus in order to run an `appd` a `dird` must be running.
3. `dcshell` is a tool that allows both local and remote execution of directory and application daemons.

4.2 Using `dcshell` to Launch DataCutter

`dcshell`, is used to start the directory daemon as follows:

```
> dcshell dird start
```

By default, an Xterm should popup with the `dird` running in it. If an Xterm does not popup, check to make sure that the `dird` is in your `$PATH`. Running a `dird` in this manner requires an X server. If you do not have an X server running, start the `dird` using the `-noxterm` option:

```
> dcshell -noxterm dird start
```

NOTE: the `dird` will be launched via an ssh remote command. There are several related issues:

- if it seems there are problems in launching a `dird`, try running with the `-verbose` and `-debug` arguments (e.g. `dcshell -verbose -debug dird start`) and look for debug output that may provide some clues as to what is happening.
- If you are launching a local `dird` (i.e. you have not set `DATA CUTTER_DIRDHOST` to point to a remote host), then this item does not apply to you. If you are launching a remote `dird`, depending on the configuration of

remote ssh version and remote shell, your remote shell's startup files may not be sourced, meaning that a remote `dird` will likely not be in your path. One way around this problem is you can create a `.datacutter.source` file in the home directory of your remote machine. That file, if it exists, will be sourced immediately before the `dird` is executed. This is an ideal place to set up a path to the remote `dird`. Note, however, that the `.datacutter.source` file must be written in a shell syntax that is compatible with your login shell on that remote machine. Here is an example `.datacutter.source`, for the ksh shell:

```
PATH=${PATH}:${HOME}/dev/nuevodc/build/bin
export PATH
```

Alternatively, you can also use a custom remote source file, with the syntax:

```
dcshell dird start -remoteDirdSrc .someRemoteFile
```

Now that `dird` is launched, we need to run application daemons on each node that we wish to distribute an application across. This can be done with `dcshell` as follows:

```
> dcshell appd start [host1] [host2] ... [host n]
```

This will create an Xterm with an `appd` running in it for each host specified. If an X server is not running on the local machine, use the `-noXterm` option with the above command to suppress the Xterm window. You will need to make sure a path to `'appd'` is available on each remote host.

NOTE: the `appd` will be launched via an `ssh` remote command. There are several related issues:

- `ssh` and `sshd` must be installed on every host involved
- If your username is different on the host where you launch the `dcshell` versus the host where you are trying to run the `appd`, you will need to configure your `ssh` to map local username to your remote username. For the OpenSSH version of `ssh`, this means you need to add some lines like the following to your `~/ .ssh/config` file on your local machine:

```
Host remotehost
  User remoteuser
```

replacing “remotehost” and “remoteuser” with the appropriate hostname and user strings, respectively.

- if it seems there are problems in launching a remote `appd`, try running with the `-verbose` and `-debug` arguments (e.g. `dcshell -verbose -debug appd start host1...`) and look for debug output that may provide some clues as to what is happening.
- Depending on the configuration of remote `ssh` version and remote shell, your remote shell's startup files may not be sourced, meaning that a remote `appd` will likely not be in your path. One way around this problem is you can create a `.datacutter.source` file in the home directory of your remote machine. That file, if it exists, will be sourced immediately before the `appd` is executed. This is an ideal place to set up a path to the remote `appd`. Note, however, that the `.datacutter.source` file must be written in a shell syntax that is compatible with your login shell on that remote machine. Here is an example `.datacutter.source`, for the ksh shell:

```
PATH=${PATH}:${HOME}/dev/nuevodc/build/bin
export PATH
```

Alternatively, you can also use a custom remote source file, with the syntax:

```
dcshell appd start -remoteAppdSrc .someRemoteFile host1 host2 ...
```

4.3 Dynamic Listen Ports

At various times, DataCutter needs to allocate a new listen port for incoming connections (the exception to this is `dird`, which uses a fixed listen port, as discussed above). DataCutter by default will listen for incoming connections in the port range 46000-50000. Thus, if you are using DataCutter on a firewalled machine, you will need to at least open incoming connections on the port range 46000-50000 on the firewall to allow communication to that node. If you don't like opening that many ports, you can try just opening as much of the low end of the range as you feel comfortable, as the ports are allocated from the low end of the range first.

If you don't like the default port range, you can override this by setting, on each machine, two environment variables (`DATA CUTTER_DYNAMICBINDPORTLOW` and `DATA CUTTER_DYNAMICBINDPORTHIGH`) to the bounds of the port range you'd prefer. These are only read through environment variables, and not via `~/ .datacutter.conf`, as they are a per-machine configuration.

4.4 Sample Application

DataCutter contains a sample application, `GridArrayAverager` which can be used to test a DataCutter installation. `GridArrayAverager` is simple application that will be used to demo some of the features of DataCutter. `GridArrayAverager` reads in a list of integers from a text file and divides the list into several buffers. Each buffer is then sent out on the grid to one of many processes, each process calculates the sum of all the integers in a buffer. The sum of each buffer is sent to another process on the grid who aggregates all the sums and calculates the average of all integers. Although this application seems like overkill for the simple problem of calculating the average, its workflow is similiar to many image understanding applications, hence it is a good example for introducing DataCutter.

A DataCutter application consists of two entities, the `filters` and the `console process`. The goal of the filters is to split the application into well defined tasks, which can then be distributed across the grid. For example `GridArrayAverager` contains three filters `ArrayReader`, `ArrayAdder`, and `ArrayAverager`. The `ArrayReader` reads in the text file and splits the integers into buffers. The `ArrayAdder` takes the sum of the integers in a buffer. The `ArrayAverager` aggregates the sum of each of the buffers and calculates the average of all integers.

The `console process` creates a layout or graph that describes the execution and distribution of the filters. For example in `GridArrayAverager`, a single `ArrayReader` filter creates buffers and sends each of them to 1 of n `ArrayAdder` filters, each of which takes the sum of a buffer and sends the sum to a single `ArrayAverager` filter who in turn aggregates the sums and calculates the average of all integers upon receiving the sum of the last integer buffer.

4.5 Running GridArrayAverager

The executable, `GridArrayAverager` is located in `DATA CUTTER_ROOT/dcbuild/bin`. `GridArrayAverager` requires two command line arguments, the shared object library containing the filters and the input file. Assuming a `dird` and at least one `appd` have been started, then `GridArrayAverager` can be run as follows (type the whole string):

```
> GridArrayAverager -lib DATA CUTTER_ROOT/dcbuild/lib/libgridArrayAveragerFilters.so
  -file DATA CUTTER_ROOT/nuevodc/quality/system/array.txt
```

The execution of `GridArrayAverager` will differ depending on the number of application daemons running. By default the `GridArrayAverager` console specifies to run one `ArrayReader` filter, two `ArrayAdder` filters, and one `ArrayAverager` filter. If there are four application daemons running one filter will be started on each `appd`. If there are less than four application daemons running, the `DataCutter` default scheduler will run multiple filters on a application daemon. The output for each filter will appear in the application daemon's `xterm` window. The application daemon running the `ArrayAverager` filter will contain an output message that displays the average of all the integers.

4.6 Shutting Down DataCutter

To exit `DataCutter`, each `appd` must be shut down and the connection to those systems must be released. Let's assume that a `dird` has been started on `host1` and it's corresponding `appd`'s has been started on `host2`, `host3`, ..., `hostn`. First, the user must terminate each `appd`. This is done with the following command executed in an `XTerm` window.

```
> dcshell appd stop host2
> dcshell appd stop host3
...
> dcshell appd stop hostn
```

This may seem unnecessary to have to stop each `appd` individually, but there are cases in which a `DataCutter` user would want to terminate a single `appd` while keeping the others running. After each `appd` has been stopped, the user must now stop the `dird`. This is done with the following command.

```
> dcshell dird stop
```

Now each `DataCutter` filter and console process has been stopped.

5 Configuration

The execution behavior of DataCutter components can be altered by specifying configuration variables. This section provides a description of the format and contents of those configuration variables. Users and/or system administrators may customize DataCutter by providing the appropriate configuration files. When a DataCutter component needs to access one of the configuration parameters it searches through the following list, in the given order. If any item in the list is missing, it is skipped. Setting any of the configuration variables in any item of the following list overrides the settings in the subsequent items. For example, a command line argument (e.g `-DATACUTTER_HOME /usr/local/DC3`) will override all configuration settings done by other means (environment variable, application configuration file, etc).

1. Command line argument,
2. Environment variables,
3. Application configuration file in your home directory `~/<appname>-datacutter.conf`,
4. Application configuration file `<appname>-datacutter.conf` in the current directory,
5. User's default DataCutter configuration file: `~/ .datacutter.conf`,
6. System-wide DataCutter configuration file: `DATACUTTER_HOME/etc/datacutter.conf`
7. Built in defaults.

Here `<appname>` is the name you provide to `Init` method `DCInstanceFactory`.

Table 2 contains a complete list of all DataCutter configuration variables.

Environment Variable	Description
DATA CUTTER_HOME	DataCutter installation directory
DATA CUTTER_DIRDHOST	Host name for dird
DATA CUTTER_DIRDPORT	Port number for dird
DATA CUTTER_COMMPROTOCOL	The communication protocol that DataCutter should use. (TCP or GSS.TCP)
DATA CUTTER_DIRDTYPE	Type of directory daemon being used. (Standalone or Globus)
DATA CUTTER_BINARYPATH	Location of datacutter binaries. (Standalone or Globus)
DATA CUTTER_USER	Username that DataCutter will run under.
DATA CUTTER_SINKQSIZE	Size of the sink queue (maximum number of unread buffers on streams)
DATA CUTTER_WORKQSIZE	Size of the work queue
DATA CUTTER_PORTWRITEPOLICY	Default write policy for output ports
DATA CUTTER_PORTREADPOLICY	Default read policy for input ports
DATA CUTTER_FILTEREXITSTATSFILE	Filename for filter statistics on exit
DATA CUTTER_REMOTESETENV	Set environment variables on remote hosts
Debugging related variables for Standalone DataCutter	
DATA CUTTER_DEBUGSRCDIR	Directory of source files for debugger
DATA CUTTER_DEBUGEXTRACMD	Commands to send to the debugger
DATA CUTTER_REMOTEX11DISPLAY	Enable automatic handling of remote DISPLAY
DATA CUTTER_WRAPPERCMD	Specify wrapper to invoke datacutter remote instances (e.g. valgrind)
DATA CUTTER_WRAPPERARGS	Specify arguments to above wrapper (e.g. -leak-check=yes -num-callers=20)

Table 2: DataCutter configuration variables

5.1 Configuration File

A DataCutter configuration file begins with the line `[DataCutter-Config]` and may contain any number of configuration variable settings. Variables are assigned values by following the name with an '=' and the value. All lines beginning with the character '#' are ignored and can be used for comments. In the configuration files; the prefix `DATA CUTTER_` should be omitted from all configuration variables. Below is an example of a datacutter configuration file:

```
[DataCutter-Config]

#DataCutter Installation Directory
HOME = /home/johndoe/projects/nuevodc

# Path to the DataCutter binaries
# These include dird dcshell appd
```

```
BINARYPATH = /home/johndoe/projects/bin/nuevodc/bin

#The host to run the directory daemon on
DIRDHOST = localhost

#The Port number the directory daemon should listen on
DIRDPORT = 6100

#The Type of dird being used
DIRDTYPE = Standalone

#The user account in which datacutter will run in
USER = johndoe

# The DataCutter Communication Protocol
# Use TCP for tcp communication
# Use GSS_TCP for globus gss communication
COMMPROTOCOL = TCP

# sizes of remote process filter queues
SINKQSIZE = 100
WORKQSIZE = 20

# Default Read and Write Policies
PORTWRITEPOLICY = RR
PORTREADPOLICY = RR
```

5.2 Directory Daemon

Every DataCutter application needs to connect to a Directory Daemon in order to query available hosts and filter implementations. Hence, configuration variables `DATA CUTTER_DIRDHOST`, `DATA CUTTER_DIRDPORT` and `DATA CUTTER_DIRDTYPE` are important configuration variables. By default `DATA CUTTER_DIRDHOST` will be set to the name of the localhost that dcshell is being executed from. The default values for `DATA CUTTER_DIRDPORT` and `DATA CUTTER_DIRDTYPE` are 6100 and Standalone.

5.3 Communication Layer

The DataCutter communication layer provides multiple protocols for network communication. In this release DataCutter allow TCP and GSS Secure TCP communication. By default DataCutter uses TCP for communication, however if DataCutter is built with the Globus toolkit, secure communication can be achieved using GSS. Details on using the Globus toolkit with DataCutter can be found in the extension section of this document.

5.4 Queue Sizes

The variables `SINKQSIZE` and `WORKQSIZE` determine the size of the remote process filter queues.

```
[DataCutter-Config]
# sizes of remote process filter queues
SINKQSIZE = 100;
WORKQSIZE = 20;
```

5.5 Stream Policies

The variable `PORTREADPOLICY` determines the policy for reading from port. There are currently three policies available:

Ordered	Reads in a round robin manner from each source filter
Unordered	Reads in FIFO order (default)
One	In each read, only the first available buffer from any source filter is read, the others are discarded

Table 3: Port Read Policy

The variable `PORTWRITEPOLICY` determines the policy for writing to multiple copy sets. There are currently three policies available:

RR	round robin per copy set (default)
WRR	weight based on the number of copies per copy set
DD	demand driven push flows based on acks

Table 4: Port Write Policy

The policy for individual streams can be set on a case by case basis in the placement file.

5.6 Development Debugging

To make development easier, there is limited (hard-coded) support for debugging. To start all remote processes in a debugger, uncomment or add the `"DEBUGSRCDIR"` line to the DataCutter configuration file. Note that only GDB is supported, and DBX support can be set by editing the DataCutter source code and changing a `#define`. The line `"DEBUGEXTRACMD"` allows for debugger-specific settings to be done before execution starts. In the example below, a breakpoint is set. This mode of debugger support assumes the current directory is a shared file system, and readable from all remote processes, due to the creation of temporary files in the current run directory for the debugger command input script.

```
[DataCutter-Config]
```

```
# enable remote process started in gdb/dbx
DEBUGSRCDIR = ../example ../src ../src/lib
DEBUGEXTRACMD = b mutex.cpp:71
```

There is limited support for just-in-time debugging (JITD). By default, DataCutter will install signal handlers for SIGKILL, SIGBUS, SIGSEGV, SIGFPE, and attempt to spawn GDB in an Xterm when an error is signalled. The Xterm implies all remote processes must be able to access the current X11 display, and have the appropriate \$DISPLAY variable set in the appd on each remote node. Due to implementation details, when the JIT debugger Xterm window appears, choosing the continue command ('c' for GDB), will usually cause the offending statement to be reached.

Another way to debug is to use the WRAPPERCMD and WRAPPERARGS variables. If WRAPPERCMD is given, the WRAPPERCMD will be placed at the beginning of the remote command line (followed by WRAPPERARGS if given), before the normal remote execution command and args. For example, on linux/x86 platforms, you can use the valgrind memory debugger to wrap remote executions by specifying WRAPPERCMD=valgrind, and optionally WRAPPERARGS=-leak-check=yes. Or, you can invoke gdb inside xterm. Because gdb cannot simply take the application arguments at the end of the gdb command line like valgrind can, the program arguments need to be massaged somewhat and passed in via an init file. As an example, the following shell script can be used as a WRAPPERCMD. Simply name it xterm-gdb-wrapper, place it in a directory in your path, and then you can set WRAPPERCMD=xterm-gdb-wrapper to cause an xterm window to pop up with a gdb session broken at the beginning of the remote executions. Here is the xterm-gdb-wrapper shell script:

```
#!/bin/sh

if [ $# -lt 1 ]; then
    echo "usage: `basename $0` <command> [args]..."
    exit 1
fi

CMD=$1
shift

CMDFILE=.debug-commands.$$
echo "set args $*" > $CMDFILE
echo "break main" >> $CMDFILE
echo "run" >> $CMDFILE
xterm -title "`basename $0` $CMD $*" -fn fixed -e gdb --command=$CMDFILE $CMD
rm -f $CMDFILE
```

This will break the remote instances inside the main() method. Normally, you will want to debug a remote invocation of a filter. So for debugging a filter FooBar's ::process method, simply do a 'b FooBar::process' and continue.

5.7 X11 Remote Display

The appd daemon can be started in one of two modes: "xterm-ssh" or "daemon", determined by the "-noxterm" option to dcshell. The former requires two local processes per remote host. The latter requires no local processes, but some debugging capability is lost (such as starting all remote processes in GDB instead of using JITD). Note: the use of the `rov.tcl` viewer can be used to monitor textual screen output from the appd and application sub-processes when daemon mode is used. Daemon mode also introduces a problem when dealing with X11 server access control, since no X11 tunnel exists to use.

Consider the following common scenario. Ssh with X11 tunneling is used to read a front end host of a parallel machine, called "front" in this example. Ssh handles the details to create a X11 connection from "front" to the user's actual workstation display. Assuming the user has started their X11 server with Xauth access control, the correct cookie is installed by ssh onto the "front" host. If we want to start appd in daemon mode, and we want to be able to display X11 windows from the back-end nodes, the cookie installed by Ssh on "front" needs to be transferred to the back-end nodes, and the `$DISPLAY` set accordingly to point to the ssh X11 tunnel on "front". Care is needed to avoid cookie conflict when a single shared file-system is used across "front" and all the back-end nodes. Given the non-negligible overhead in adding the remote cookies in the required way, dcshell supports this `$DISPLAY` transfer when starting appds.

The following configuration options control the remote X11 display:

```
# enable automatic handling of remote $DISPLAY, so remote
# processes can open windows on the local user's screen.
# (omit) - do nothing; ssh/user login will set everything
# (this will not work with appds in daemon mode)
# display - forward $DISPLAY to remote hosts
# xauth - forward $DISPLAY; forward console xauth cookie
remote_x11_display = xauth
<remotehost>:DataCutter-2.1/etc/appd.cfg
# forward console process $DISPLAY setting (default = true)
REMOTEX11DISPLAY = xauth list-add
```

To run in daemon mode:

1. (ssh into front-end host and insure `$DISPLAY` is correct)
2. front> dcshell dird start
3. front> dcshell -noxterm appd start host1 host2...
4. front> (run applications)

To run in xterm node:

1. (ssh into front-end host and insure `$DISPLAY` is correct)

2. front> dcshell dird start
3. front> dcshell appd start host1 host2...
4. front> (run applications)

5.8 Build Options

The DataCutter extensions that are distributed with the DataCutter toolkit are not built by default. However they can easily be enabled by running CMake interactively:

```
> cd DATACUTTER_ROOT/dcbuild
> cmake -i ../nuevodc
```

Interactive CMake will ask whether or not to enable an option. To enable an option type “true”, to disable an option type “false”. Table 5 contains a list of the DataCutter options:

Option Name	Description
BUILD_DOCS	Do you wish to build documents?
BUILD_TESTING	Build the testing tree
DART_ROOT	If you have Dart installed, where is it located?
USE_XML	See the XML extension section 8.1
USE_GLOBUS	See the Globus extension section 8.2
USE_SRB	See the SRB extension section 8.3
USE_NWS	See the NWS extension section 8.4

Table 5: Build Options

It is important to note that enabling some options will require entering additional information. For these cases see the extensions section for that particular option for more detail. Running CMake interactively will prompt for other options other than the ones listed above, for those options just hit the “enter” key.

6 Application Development

This section provides an introduction to the DataCutter application programming API which allows users to define custom applications. This section will present central ideas to the coding and compilation process in the context of the example `GridArrayAverager` (`GridArrayAverager.cpp`).

This simple application calculates the average of a set of 10000 integers using three filters:

- The "Reader" filter which reads the numbers from the file `array.txt`
- The "Adder" filter which sums the numbers read by the Reader
- The "Averager" filter which calculates the average of the data

The filters are set up to use a system of DataCutter Pipes in order to pass data from one filter to another. This example only covers part of the functionality of the DataCutter library. See the class header files for a more complete description.

A DataCutter application has a work-flow that must be defined during creation. Applications, which are often referred to as console processes, must specify the layout (what filters the application will use, and how they are connected). The application can optionally define a placement (mapping of filter to hostname) of the filters as well.

A placement is the description of how the console uses the hosts to run its filters. For example, a user could specify that one particular filter is to be run on a particular host, and all remaining filters are to be assigned to the hosts by the DataCutter scheduler. This is useful because for some filter processes, running them on multiple hosts offer no improvement. For many other processes, running many simultaneous filters running across a network can drastically improve performance.

This presents us with the idea of *transparent copies*. When a single DataCutter filter is run across multiple computers, the DataCutter main process treats them as a single process. The DataCutter scheduler controls what information needs to be sent where and what to do with the output. The multiple filters are transparent to the main process. It doesn't matter how many filters there are, they will all be handled as a single filter and the scheduler will arrange the work for the user. For the example, the Adder is the filter that would have transparent copies.

6.1 Creating a simple application

The complete source for `GridArrayAverager` is in `/DATACUTTER_ROOT/nuevodc/quality/system`

Before continuing, it is necessary to understand how DataCutter executes. We must create a process to load the data into a buffer, then send the buffer to the filters for the work to be done. This process is the console application. It manages all of the filter creation and output handling. For the `GridArrayAverager` sample application, there are 3 filters to be created. The console application creates these filters from the `GridArrayAverager` library and assigns them work to do.

File Name	Description
ArrayReader.cpp	The filter process that read the integers from input file.
ArrayAdder.cpp	The filter process that adds the integers received from the reader.
ArrayAverager.cpp	The filter process that calculates the average.
GridArrayAverager.cpp	The console process that gives work to the filters.
GridArrayAveragerFactory.cpp	The factory class used to load the library.
array.txt	The integers used as the input.

Table 6: GridArrayAverager source files

To begin, we must initialize the factories. The filter factory is needed to register the libraries used for the filters. The instance factory is required to begin an instance of DataCutter itself.

```
DCInstanceFactory* DC = DCInstanceFactory::New();
DCFilterFactory* dfac = new DCFilterFactory();
```

Now we need to register the library containing the filters that this application will need to run. For this example, the location of the filter library is specified in the command line. More information on creating filter libraries may found in the Filter Development section of this document.

```
dfac->registerLibrary(argv[2])
```

DataCutter is initialized by passing in the application name and command line arguments to the instance factory `init()` function:

```
DC->Init("GridArrayAverager", dfac, &argc, &argv)
```

Optionally, you can set up a filter error handler so you can be aware from the console process that a filter returned an error from its `::initializeWork`, `::process` or `::finalizeWork` methods. For example:

```
DC->registerFilterErrorHandler(myErrorHandler);
```

where `myErrorHandler` might be defined as follows:

```
void myErrorHandler(
    char * filterName,
    int instanceNum,
    char * hostName)
{
    char cmd[ARG_MAX];
    snprintf(cmd, sizeof(cmd),
```

```

        "xmessage 'filter %s (instance %d) failed on host %s'",
        filterName, instanceNum, hostName);
    system(cmd);
}

```

so you get an X11 popup message whenever one of the filters failed.

Next, the filters used by the application need to be created from the filter factory. The filter factory method, called `createFilter()`, is called to create an instance of each filter that the application will use. It will do this by using the factory method that exists in the library(s) that were registered with the filter factory. The first argument to `createFilter()` is the name that you want to associate with that particular instance of the filter. The second argument is the type of filter of which you would like to create an instance.

```

DCFilter *reader=dfac->createFilter("reader", "ArrayReader");
DCFilter *adder=dfac->createFilter("adder", "ArrayAdder");
DCFilter *averager=dfac->createFilter("averager", "ArrayAverager");

```

To create the execution work flow we must create a layout. The `DCLayout` API allows you to generate a logical layout of filters. The `DCPipe` API is used to logically connect the output ports to input ports of any two filter instances. In this example you can see that we connect the output of the reader to the input of the adder, and the output of the adder to the input of the averager.

```

DCLayout* layout = new DCLayout();
layout->addFilter(reader);
layout->addFilter(adder);
layout->addFilter(averager);
layout->addPipe(reader->getOutPort("out"), adder->getInPort("in"));
layout->addPipe(adder->getOutPort("out"), averager->getInPort("in"));

```

`GridArrayAverager` creates an optional placement because multiple copies of the `ArrayAdder` filter can be run in parallel. Doing this distributes the addition process to multiple nodes, thus increasing the throughput of the application. For this example two copies of the `ArrayAdder` filter are created. In this case the `DataCutter` scheduler decides which nodes to place those copies on, however the placement API allows the specification of specific hosts. When the `ArrayReader` sends a buffer out to be added up, `DataCutter` decides which of the `ArrayAdder` filters to send the buffer to for processing. (See `Stream Policies`). When a `ArrayAdder` sends there output to the `ArrayAverager`, the `DataCutter` runtime system manages the handling of the buffers, from the `ArrayAverager`'s point of view it input is coming from one place.

```

DCPlacement* placement = new DCPlacement(layout);
placement->add(adder, 2);

```

Having specified the layout and placement, it is now possible to create an instance of the filter group and assign it work. An instance is of type `DCInstance` and it is created with a call to the `NewInstance` function of the

`DCInstanceFactory`. This will initialize an instance using the placement that you provide it. This initialization takes care of checking the logical layout of the filters, verifies the placement is acceptable, and places any of the filters that were not placed. After this step the instance is ready and waiting for work to be appended to it.

```
DCInstance* instance;  
DC->NewInstance(instance, placement);
```

A `work buffer` is a buffer containing some specific data required to get a particular process done. The work buffer can be used to store and send any information to all filters. The Work Buffer broadcast to all filters before any of the filter execute. Each filter will get a handle to it on the remote machines and can read what is in the buffer. A common use for this is to be able to send a description of what the filters should do or how to do what they will do. In the `GridArrayAverager` example the work buffer contains the location of the input file, which is need by the `ArrayReader`.

```
DCWork work;  
work.buf.Empty();  
int length = strlen(argv[4]);  
work.buf.Append(length);  
work.buf.Append(argv[4], length);
```

By writing to the buffer contained within the previously created work object, it then possible to define a unit of work and send it to the filter group for processing. Sending work to a filter group is accomplished using the `AppendWork` function of the `DCInstance` object.

```
instance->AppendWork(work)
```

It is important to note that you could also append different work buffers here if you wanted to use the same filter layout and placement to work on many different sets of data, or on the same data with different configuration parameters.

Before assigning additional work to one or more filters, the application can wait for the completion of a particular unit of work via the `WaitWork` call or wait for the completion of any unit of work using `WaitAnyWork`. Negative return values from these two functions indicate an error, two of which correspond to values returned by the filters themselves. A filter which returns `DC_RTN_CONT` will cause `Wait(Any)Work` to return `DC_ERR_WorkFail`, the interpretation of which is application specific. It is the responsibility of the filter designer to indicate whether or not part or all of the results for the unit of work should be discarded and whether or not that particular unit can be retried. The meaning of a filter which returns `DC_RTN_ABRT`, however, is unambiguous. This will result in the destruction of the filter, the closing of its input and output streams, and the return of `DC_ERR_InstanceFail` to `Wait(Any)Work`. In this situation, the filter group instance is corrupt and will need to be shut down with a call to `StopInstance`.

```
DC->WaitWork(wh);
```

Note: All error checking has been removed to make this definition simpler. To see what is missing, check the source files.

6.2 Include Files

There is one include file which contains includes for most classes that may be needed by the application developer. All of the following class and function definitions generally used by an application developer are contained in the single header file `DataCutterApplicationDeveloper.h` which must be referenced with:

```
#include <DataCutterApplicationDeveloper.h>
```

6.3 Error Reporting

Return codes from class library calls can be converted to human readable error messages with:

```
DC_strerror(int)
```

6.4 Compilation

The easiest way to build an application with DataCutter is to build your new application with CMake and include the `UseDataCutter.cmake` file that is generated in the DataCutter build directory. This will automatically allow your application to inherit any flags or libraries needed to build a DataCutter application. Compilation requires use of the DataCutter library `libdc.a`. To include `libdc.a` in your CMake build just add the text `LINK_LIBRARIES(dc)` after the line `INCLUDE(${DATACUTTER_BINARY_DIR}/UseDataCutter.cmake)`.

7 Filter Development

7.1 Filter Creation

Filters are created by sub-classing the abstract base class `DCFilterImplementation` and defining the three callback functions used for filter initialization, data processing, and cleanup. A logical description of interconnected filters is called a filter group. At run time, an instance of a filter group is created which processes data in units of work as determined by the main part of the application. When a new unit of work arrives, the runtime system calls the `initializeWork`, `process`, and `finalizeWork` functions in succession. The filter developer must at least implement the `process` function. At run time, each filter group may contain more than one transparent copy of any filter. These transparent copies execute collectively in parallel to process work within the filter group instance. Within a filter group instance, all the transparent copies of a particular filter on a single host is called a copy set. Filters communicate via streams which deliver buffer objects per call.

7.1.1 Buffers and Work

The `DCBuffer` is the unit of transfer for streams. This class is simply a container for a chunk of contiguous memory, which can be statically or dynamically allocated. These buffers have a maximum size and keep track of how much of that maximum is occupied. Data that will fit into the unused portion of the buffer can be appended. Extraction occurs from the head of the buffer, maintaining an extraction pointer for multiple extraction operations. An application can also just get a pointer and length of the data in the buffer, and use it in place.

When finished with a buffer that has been read from stream the `consume` method should be called to “release” the buffer, which will free the space, if the buffer is allocated from heap. When a non-heap instance of `DCBuffer` is created, `consume` flag should be set `false` (by calling `setConsume(false)`) in order to avoid accidental deletion of the instance. (*NOTE: the Work buffer, which is broadcast to all filters at the beginning of execution, should NEVER be consumed in this manner*). Table 7 shows example code for performing some common tasks with `DCBuffer`.

Activity	Example
Allocate memory	<pre>const int setElements = 10; int bufSize; DCBuffer buf; bufSize = setElements * sizeof(int); buf.New(bufSize);</pre>
Set the amount of the buffer to be used	<pre>buf.setSize(bufSize);</pre>
Get a pointer to the beginning of the buffer	<pre>char *p= (char*)buf.getPtr();</pre>
Get the size of the current buffer	<pre>for (unsigned int i = 0; i < buf.getSize(); ++i) { : }</pre>
Clear the buffer	<pre>buf.Empty();</pre>
Append a character	<pre>buf.Append(setElements);</pre>
Extract a character	<pre>buf.Append(&setElements);</pre>

Table 7: Common Tasks Using Buffers

The `DCWork` is the definition of a particular processing job or query. It is a container class for data objects `DCBuffer buf`, which allows the parameterization of the work request to be passed to the filter group, and `int wWorkNum`, which allows the unit to be distinguished from other simultaneous work requests. This information is provided to each filter on initialization.

7.1.2 Initialization

The `initializeWork` function is called when the filter is given a new unit of work. It allows the filter to discover its location in terms of its filter group and copy set, to determine the parameters of the work request, and to set up any needed resources. The prototype for the initialization function is:

```
DC_RTN_t initializeWork(DCFilterInitArg&)
```

The argument of type `DCFilterInitArg` contains data items `char *sbFilterName`, which provides the name of the filter, and `DCWork *pwork`, which gives the work description. Accessing these items to retrieve the work description can be achieved in the following fashion:

```
DC_RTN_t initializeWork(DCFilterInitArg &initarg) {
    char* outfile = initarg.pwork->buf.getPtr();
    :
    return DC_RTN_OK;
}
```

The return type `DC_RTN_t` can have three values: `DC_RTN_OK`, indicating that the processing for this unit of work has finished and that it is ready for another unit, `DC_RTN_CONT`, indicating an error occurred with this unit of work, but that the filter can continue processing, and `DC_RTN_ABRT`, indicating a critical error requiring the death of the filter and ultimately the destruction of the entire filter group instance.

For the `GridArrayAverager` example, a unit of work is defined in terms of the filename in which the integers that will be average are contained. Thus only the `ArrayReader` class is concerned with the contents of this buffer, which it uses for partitioning the integers into smaller buffers such that the data can be distributed to transparent copies of the `ArrayAdder` filter for summation. To summarize the ideas involved in defining filters, creating buffers, and initializing a filter in the context of this example, portions of the three filter class definitions follow.

The `ArrayReader` defines the size of the integer buffers it will send to `ArrayAdder` class:

```
DC_RTN_t ArrayReader::initializeWork(DCFilterInitArg &initarg) {
    cout<<"[Array Reader] - Init Called"<<endl;
    block_size = 100;
    return DC_RTN_OK;
}
```

The `ArrayAdder` class needs to create a buffer in order to return its results to the `ArrayAverager` class, so its `initializeWork` function is:

```
DC_RTN_t ArrayAdder::initializeWork(DCFilterInitArg &initarg) {
    cout<<"Array Adder Init Called"<<endl;
    bufSize = 2 * sizeof(int);
    buf.New(bufSize);
    buf.setSize(bufSize);
    return DC_RTN_OK;
}
```

The `ArrayAverager` does not need a `initializeWork()` method.

7.1.3 Processing

Once `initializeWork` has returned, the `process` function is called. This is the stage in which the actual "work" is performed on the data. The prototype for the processing function is:

```
DC_RTN_t process(DCFilterArg&)
```

The argument of type `DCFilterArg` is derived from `DCFilterInitArg` and provides the previously mentioned data items in addition to handles for the input and output streams. These are accessed by the arrays `ins` and `outs`

which are of size `nins` and `nouts`, respectively. The functions `insIndex` and `outsIndex` provide for stream name to index translation. An input stream is of type `DC_PipeInStream_t` and supports a `read` function, and an output stream is of type `DC_PipeOutStream_t` and supports two write functions, `write` and `write_nocopy`.

The difference between `write` and `write_nocopy` is that `write` does a deep copy of the buffer object and memory region as needed to allow the caller to modify the buffer immediately after this call returns. For example, a co-located sink will cause the buffer object and memory region to be duplicated and placed directly in the consumer's queue. However, `write_nocopy` uses the given buffer. If co-located with the sink this will be deposited directly in that queue. Use of stack allocated `DCBuffer` objects for this call is not recommended due to undesirable results if the stack frame is removed before the object is dequeued/used by a co-located consumer.

Activity	Example
Get an input stream index by name	<code>int inIndex = arg.insIndex("P-S");</code>
Get an output stream index by name	<code>int outIndex = arg.outsIndex("S-C");</code>
Read from the input stream and write to the output stream	<pre>DCBuffer *pbuf; while ((pbuf = arg.ins[inIndex].read())) { : // process input if (arg.outs[outIndex].write(pbuf)!=DC_ERR_OK){ cout << arg.sbFilterName << ": failed write on buffer " << pbuf << endl; } pbuf->consume(); }</pre>
Complete processing	<code>return DC_RTN_OK;</code>

Table 8: Working with streams

The `ArrayReader` reads in a list of integers from a file and partitions the list into buffers. Each buffer is then sent to a transparent copy of the `ArrayAdder` filter.

```
DC_RTN_t ArrayReader::process(DCFilterArg &arg){
    DCWork *work = arg.pwork;
    int length;
    work->buf.Extract(&length);
    char *array_file = new char[length+1];
    work->buf.Extract(array_file,length);
    array_file[length] = '\0';
    ins.open(array_file);
    int count=0;
    int *array = new int[block_size];
    int outId = 0;
    int num;
    while(ins >> num){
        array[count]=num;
    }
}
```

```

count++;
if(count==block_size){
    cout<<"[ArrayReader] - Read in "<<count
        <<" integers and sent them off to be averaged."
        <<endl;
    DCBuffer buf = get_buffer(array,count);

    if (arg.outs[outId].write(&buf) != DC_ERR_OK) {
        cerr << arg.sbFilterName << ": failed write to buffer" << endl;}
    count=0;}
}
if(count>0){
    cout<<"[ArrayReader] - Read in "
        <<count
        <<" integers and sent them off to be averaged."
        <<endl;
    DCBuffer buf = get_buffer(array,count);
    if (arg.outs[outId].write(&buf) != DC_ERR_OK) {
        cerr << arg.sbFilterName << ": failed write to buffer" << endl; }
}
ins.close();
cout<<"Process Call Finished"<<endl;
delete array;
delete array_file;
return DC_RTN_OK;
}

```

In GridArrayAverager the ArrayAdder's task during the process stage is to take the sum of each buffer of integers it receives. It then sends the sum of each buffer to the ArrayAverager filter.

```

DC_RTN_t ArrayAdder::process(DCFilterArg &arg){
    DCBuffer *pbuf;
    // Get the input stream index
    int inId = 0;
    // Get the output stream index
    int outId = 0;
    while ((pbuf = arg.ins[inId].read())) {
        int elements=0;
        int sum = 0;
        int *in_buf = (int*)pbuf->getPtr();
        int *out_buf = (int*)buf.getPtr();
        for(int i = 0; i<(int)(pbuf->getSize()/sizeof(int)); i++){
            sum = sum+in_buf[i];
            elements = elements+1;
        }
    }
}

```

```

    out_buf[0] = elements;
    out_buf[1] = sum;
    cout<<"[ArrayAdder] - Received buffer, added "
         <<elements
         <<" integers to a total of "
         <<sum
         <<endl;
    if (arg.outs[outId].write(&buf) != DC_ERR_OK) {
        cout << arg.sbFilterName << ": failed write on buffer"<< endl;
    }
    pbuf->consume();
}
cout<<"Array Adder Process Done"<<endl;
return DC_RTN_OK;
}

```

In the process stage, the ArrayAverager filter receives buffers from the ArrayAdders, containing the sum of a partition of integers. Upon receiving all the buffers the ArrayAverager computes the average.

```

DC_RTN_t ArrayAverager::process(DCFilterArg &arg){
    DCBuffer *pbuf;
    // Get the input stream index
    int inId = 0;
    int average = 0;
    int sum = 0;
    int elements = 0;
    while ((pbuf = arg.ins[inId].read())) {
        int *in_buf;
        in_buf= (int*) pbuf->getPtr();
        elements = elements+in_buf[0];
        sum = sum + in_buf[1];
        cout<<"[ArrayAverager] - Received a buffer current sum of "
             <<elements
             <<" integers is "
             <<sum<<endl;
        pbuf->consume();
    }
    average= sum / elements;
    cout<<"[ArrayAverager] - Determined the average of all integers to be: "
         <<average<<endl;
    return DC_RTN_OK;
}

```

7.1.4 Finalization

After the `process` function has returned, the `finalizeWork` function is called. This stage of the filter can be used for freeing allocated memory, closing file streams, or any other type of clean up required. The prototype for the finalization function is:

```
DC_RTN_t finalizeWork(void)
```

For all three filters in `GridArrayAverager`, there is nothing to be done during this stage, so the function just returns:

```
DC_RTN_t finalizeWork(void) {  
    return DC_RTN_OK;  
};
```

If there were a case of a filter needing to close its initialization, this is essentially the same as a destructor. All allocated memory must be returned to the operating systems control.

7.2 Registering Filters

The main part of the application specifies the number and configuration of the defined filters and turns over control to the DataCutter run time system. The run time system then makes the calls for initialization, processing, and cleaning up. The main part of the application also defines what work is to be done and assigns this work to particular filter groups. As such, it can provide an interface to an external query generating client or be a stand-alone application working on internally defined tasks. The actual instantiation of the various filters is handled by a helper function which must be provided during system initialization. The prototype for this function is:

```
DCFilterImplementation *filter_factory(char *)
```

The run time system is initialized by creating an instance of the `DCFilterFactoryImpl` class and calling its `init` function. If the `remoteConsoleType` parameter to `init` is set to `CUSTOM` this application will run on each host running filters, it must then be determined whether or not this particular instance is the console process with a call to `isRemoteProcess`, and if such is the case, the process prepares itself to run filter code by calling `RemoteProcess`. If the `remoteConsoleType` parameter to `init` is set to `GENERIC` then the check for `isRemoteProcess` does not need to exist in the console code.

For the `GridArrayAverager`, three filters are defined, `Reader`, `Adder`, and `Averager`. For the run-time system to be able to generate these filters when needed, the following function is defined:

```
DCFilterLibrary *dcLoadFilterFactory()  
{
```

Activity	Example
Instantiate and initialize the run time system	<pre data-bbox="711 411 1542 600">DCFilterFactoryImpl DC; int remoteConsoleType = DCREMOTECONSOLETYPE_GENERIC; if (DC.init("GridArrayAverager", &dfac, &argc, &argv, remoteConsoleType)) { exit(1); }</pre>
If remoteConsoleType is set to DCREMOTECONSOLETYPE_CUSTOM, check whether the current process is remote, and if so, hand over control to the run time system.	<pre data-bbox="711 680 1144 802">if (DC.isRemoteProcess()) { DC.RemoteProcess(); return 0; }</pre>

Table 9: Tasks involved in bringing up the DataCutter run time system

```
DCFilterLibrary *lib = new DCFilterLibrary("DistributedArrayAverager");
lib->registerFilter(new DCFilterSpec("ArrayReader",
                                   DC_CreateFilter<ArrayReader>, 0, 1));
lib->registerFilter(new DCFilterSpec("ArrayAdder",
                                   DC_CreateFilter<ArrayAdder>, 1, 1));
lib->registerFilter(new DCFilterSpec("ArrayAverager",
                                   DC_CreateFilter<ArrayAverager>, 1, 0));

return lib;
}
```

Once defined, initialization of the run-time system can occur and the process can check whether or not it was started remotely:

```
if (DC->Init("GridArrayAverager", dfac, &argc, &argv)) {
    cerr << "DC.init failed!" << endl;
    exit(1);
}
```

The task of the console process is to then:

1. Instantiate an object of type DCWork for work unit definition
2. Determine the filter layout and placement
3. Instantiate this filter group

4. Assign work

A work object is of type `DCWork` and an object is created by:

```
DCWork work;
```

8 DataCutter Extension Packages

The DataCutter extensions provide useful utilities and integration with other toolkits to enhance the functionality of DataCutter. This section will provide the details on how to setup and use these extensions.

8.1 XML Process Models and Filter Configuration

Xerces is the XML reader supported by DataCutter. The version supported by DataCutter is Xerces C2.2 The options for configuring it during installation are:

Variable Name: USE_XML

Description: Do you wish to build DataCutter with XML library support?
This will require the Xerces Library version 2.2

Variable Name: XML_INSTALL_PATH

Description: What is the path where the file . can be found

You can find out more about Xerces at <http://xml.apache.org/>

The XML extensions provide three main functionalities that can be used together or individually. The main use is for describing the logical layout and placement of filters in an XML document. The next, is to be able to list which libraries are required to run the filters listed in the layout. The third, and probably the most useful, is the ability to write filters that can be configured using XML parameter blocks. The XML parameter blocks are automatically parsed and the variables are available to the filter developer through a simple API that can be found in the `DCXMLBasicFilter`.

8.1.1 XML for describing filter libraries

With the USE_XML option on DataCutter will build a set of extensions that enable the creation of an XML description of the libraries to be used. The `DCDynamicFactory` class will be used to handle parsing the XML description of libraries and will load those libraries into the generic console.

```
<libraries>
  <library>
    <name>test_filters</name>
    <linux path="libtestFilters.so"/>
  </library>
</libraries>
```

8.1.2 XML for modeling processes

With the USE_XML option on DataCutter will build a set of extensions that allow for using XML descriptions of filter layout and placement, and generic consoles that can use them to run applications. DCXMLParser can take in a buffer or a file and parse it to return a DCPlacement which will have the layout of the filters and there placements if so specified. Below is an example of writing the testBuffers test using an XML model and using the DCXMLGenericConsole instead of writing a console application. Below is an example XML model for running testBuffers.

```
<processes>
  <process name='poet' type='Poet' nCopies='1'>
<!--
  Could place some of the copies of the copies on a named host by adding
  a line like this
    <placement host='hostname' nCopies='1' />
-->
    <process name='muse' type='Muse' nCopies='1'>
<!--
  Could place some of the copies of the copies on some host by adding
  a line like this
    <placement nCopies='1' />
-->
    </process>
  </process>
</processes>
```

You can combine these two sections together into one XML document and give that document to the generic console. The XML snippet below shows the document for running testBuffers. It has two main sections: libraries and processes. The libraries section is optional, however, it must contain a processes section. This document show that there is a filter “muse” of type Muse that performs some operation then send it’s data to the filter “poet” of type Poet. These filters both will come from the library libtestFilters. The xmlGenericConsole will parse this document, register the libraries, generate the layout and placement, and execute it in the DataCutter environment.

```
<job id='1'>
<libraries>
  <library>
    <name>test_filters</name>
    <linux path="libtestFilters.so"/>
  </library>
</libraries>
<processes>
  <process name='poet' type='Poet' nCopies='1'>
  <process name='muse' type='Muse' nCopies='1'>
  </process>
```

```

    </process>
  </processes>
</job>

```

The generic console, `xmlGenericConsole`, can be used to run the applications defined in the XML document. For example:

```

xmlGenericConsole <XML process model (may include the libraries)>
    <Work (may be xml)> [<shared libraries>]

```

Where `work` is the file containing the work that this instance of the application will use.

8.1.3 XML for configuring filters

With the `USE_XML` option on `DataCutter` will build a set of extensions that allow using XML configuration blocks for configuring units of work for a particular application. To use this feature the filter developer must extend the class `XMLBasicFilter`—when writing the filters. This will enable them to have access to their parameters through an API interface. For example, let's look at a sample filter that has the ability to read image files in from the file system and stream them out to the next filter. Let's say that there were two things that this filter needs for configuring itself: filename and filetype. We could use the following parameter block to configure this filter:

```

<params>
  <param value='PNG' name='filetype' />
  <param value='/home/hastings/test.png' name='filename' />
</params>

```

I could now use this XML to generate a work document for the filters that I will run. Let's say that I want to use the image read and image render filters to read in a png and stream it to a filter that can show it. I may use the following xml model for describing the layout:

```

<job id='1'>
  <processes>
    <process name='renderer' type='Renderer' nCopies='1'>
      <process name='reader' type='Reader' nCopies='1'>
        </process>
      </process>
    </processes>
  </job>

```

And the following XML work file to configure the filters that are being used in this particular instance that I want to run:

```

<processes>
  <process name='renderer' type='Renderer'>
  </process>

  <process name='reader' type='Reader'>
    <params>
      <param value='PNG' name='filetype' />
      <param value='/home/hastings/test.png' name='filename' />
    </params>
  </process>
</processes>

```

Now I can use the XML model file and work file to run an instance of this application with the parameters specified. I would simply run the following command:

```
xmlGenericConsole <XML process model> <XML Work>
```

The XML work file will be read in and broadcast to all filter instances being used for this instance. The filter instances will parse the XML work file and locate the parameter block for there filter and provide an API to get to the variables needed for that filter. Accessing these variables in any of the functions of the “reader” filter code would look like the following for example:

```

char* fileType = getParamValue("filetype");
char* fileName = getParamValue("filename");

```

8.2 Globus

The Globus Project is developing the fundamental technologies needed to build computational grids. DataCutter uses the Globus toolkit to provide enhanced functionality. These enhancements include secure communication, alternative remote process execution, and alternative directory services. This section will explain how to enable and use the Globus extensions with DataCutter.

8.2.1 Enabling Globus Extensions

To enable the Globus Extensions, DataCutter must be built and linked against specific flavors of the Globus toolkit. In order to do so the Globus Toolkit 2.2.2 (<http://www.globus.org/>) must be installed on the system with the required flavors. The DataCutter Globus extension utilizes globus_io, GRAM, MDS, and requires that they be built using a threaded flavor (ie gcc32dbgpthr).

Globus can be enabled by setting the value of the CMake option USE_GLOBUS, to true. Enabling this option will require specifying the path to the globus installation directory (GLOBUS_PATH CMake option.) Once enabled rebuild

the DataCutter toolkit using `make`, this will build the Globus/DataCutter extension code. It is important to note that in order to use MDS, one will also have to have to set the value of the CMake option `USE_MDS` to true.

8.2.2 User Certificates

Several of the following operations require that a valid globus proxy has been established, in order to use Globus' secure communication. To establish a valid globus proxy, you first need to obtain a user certificate. Details about how to do so are outside the scope of this document, but once your user certificate has been created, you should initialize a globus proxy by running `grid-proxy-init`. You can verify that your globus proxy has been initiated by running `grid-proxy-info`. Globus proxies, once established, will remain for a limited amount of time (typically 12 hours). You can manually destroy a proxy before this time has expired by running `grid-proxy-destroy`.

8.2.3 Using GRAM to Start DataCutter Daemons

DataCutter provides the ability to start up both directory daemons and application daemons using GRAM. This can be done by specifying the `-gram` option in the command line when using `dcshell`.

```
dcshell -gram dird start
dcshell -gram appd start host1 host2 .... hostN
```

It is important to note that using GRAM requires a valid globus proxy; if one does not exist the daemons will not start. When using GRAM the output of the individual daemons will be written to log files in the users home directory; the log files are named `dird-HOSTNAME-datacutter.log` and `appd-HOSTNAME-datacutter.log`.

8.2.4 Secure Communication with GSS

By default DataCutter uses plain TCP for communication between components. The Globus Toolkit uses the Grid Security Infrastructure (GSI) for enabling secure authentication and communication over an open network. DataCutter uses the Globus Toolkit's implementation of GSI, Generic Security Service API (GSS) in order to provide a method of secure communication between DataCutter components. GSS communication can be enabled by setting the DataCutter configuration variable, `DATA CUTTER_COMMPROTOCOL` to `GSS_TCP`. Communicating using GSS requires a valid globus proxy and that GRAM is used to start the directory daemons and all application daemons.

8.2.5 MDS

DataCutter provides an internal directory service interface that abstracts the communication between the DataCutter runtime system and the directory daemon. This abstraction allows datacutter to support multiple directory daemon types. DataCutter is distributed with a MDS Directory Client that implements this interface. This allows DataCutter to

use Globus's MDS as a directory daemon. The MDS directory client can be built by setting both the USE_GLOBUS and USE_MDS CMake options to true. Once built the MDS Directory Daemon can be used by setting the configuration variable, DATACUTTER_DIRDTYPE equal to 'Globus' (See Configuration Section).

8.3 Storage Resource Broker (SRB)

DataCutter supports SRB 2.0, a storage resource system. The DC-SRB extension will be configured automatically by CMake if the user selects the USE_SRB option and provides the location of the SRB installation. The launchDCGC script is installed automatically.

Variable Name: USE_SRB
Description: Use SRB?

Variable Name: SRB_INSTALL_PATH
Description: the path to the SRB installation

8.3.1 SRB Examples

Start up the DC directory daemon (dird) and application daemons (appds) if they have not already been started for you by SRB or the system administrator.

```
% dcshell dird start
% dcshell appd start akron dayton
```

Try to execute the first example application. First, add something to your SRB directory using the Scommands. Then, try to read it back using the DC SRB read filters. This example will attempt to read from an SRB object and dump the results to stdout.

```
% cd example1
% Sinit; Sput example1.c example1.c; Sls -l; Sexit
% ./example1 /home/dc.npaci example1.c
```

For the second example application, we need to do some additional work to setup the RTree index and data files. These files will be used for example2 and example3. Assuming your home directory/collection in SRB is /home/dc.npaci, execute these commands:

```
% Sinit
% Scd /home/dc.npaci
% Smkdir data
% Scd data
```

```

% cd example2/data
% Sput scene0 scene0
% Sput scenel scenel
% Sput scene2 scene2
% Sput scene3 scene3
% cd ../index-files

```

Now modify the ASCII dataset catalog file to the appropriate SRB collection path and create the binary linear index files:

```

% vi data.txt
% ../utility/create_catalogs -d data.txt
% ../utility/create_catalogs -i index.txt
% ../utility/create_catalogs -l scene0_idx.txt scene0_idx.bin
% ../utility/create_catalogs -l scenel_idx.txt scenel_idx.bin
% ../utility/create_catalogs -l scene2_idx.txt scene2_idx.bin
% ../utility/create_catalogs -l scene3_idx.txt scene3_idx.bin

```

Next let's copy the catalog and index files into SRB:

```

% Scd /home/dc.npaci
% Smkdir linear-index
% Scd linear-index
% Sput data.cat data.cat
% Sput index.cat index.cat
% Sput scene0_idx.bin scene0_idx.bin
% Sput scenel_idx.bin scenel_idx.bin
% Sput scene2_idx.bin scene2_idx.bin
% Sput scene3_idx.bin scene3_idx.bin

```

Finally, let's create the index using the DC SRB RTree Index filter (the resource in this example is unix-ohio-state):

```

% cd ..
% ../example2 /home/dc.npaci/linear-index /home/dc.npaci/data-index unix-ohio-state

```

Once this is complete, you should see a new collection (data-index) has been created:

```

% Scd /home/dc.npaci
% Sls
% Sls data-index

```

With the catalog and index created and loaded into SRB, we can run example3 to use the DC SRB RTree query filter. Here we are querying for the range in x from 0 to 1151 and the range in y from 0 to 899. The result will give us the list of data files, including offsets and buffer sizes, that we would need to read in order to fulfill the query.

```
% cd ../example3
% ./example3 /home/dc.npaci/data-index 0,0,1151,899
```

With these examples, you should be able to see some uses of the DC SRB API as well as see a couple of useful DC SRB filters that are already built for your reuse.

8.3.2 About the Files

client:

dc_api_console.h - API given to SRB client to allow for developer to use DC functionality within SRB; gives ability to create the generic console, set the layout, placement and work information as well as receive results back after the work has been completed

example[1-3] - simple, sample SRB clients that uses the DC console api to read SRB data; see above

server:

launchDCGC - needs to be put into `\$SRB_HOME/bin/commands/` directory; this program is called by SRB server when SRB client creates a DC console

dcgc - DC generic console capable of communicating with SRB client for layout, placement, work and results information; capable of loading shared libraries for filters

other:

dc_api_console.c - implementation of dc_api_console.h

dc_api_util.* - common code used by dcgc and dc_api_console

srb_filters.* - implementation of some reusable DC SRB filters to provide component for reading and indexing SRB objects

Makefile - mechanism to build both dcgc and example clients

You can find out more about SRB at <http://www.npaci.edu/DICE/SRB/>

8.4 Network Weather Service (NWS)

DataCutter supports NWS 2.2.1 as a source of network metrics. To build this extension, select USE_NWS and specify the NWS installation's location. If you also select USE_MDS, communications with NWS will use the LDAP protocol instead of the native protocol.

Variable Name: USE_NWS
Description: Use NWS?

Variable Name: NWS_INSTALL_PATH
Description: the path to the NWS installation

The NWS extension provides two extensions to DataCutter; the first is the ability to use the NWS to obtain network metrics and information about hosts, the second is a basic DCNWSscheduler which can utilize this information at runtime to place filters on hosts more efficiently. (Selecting USE_NWS enables both of these extensions at runtime as well.)

The following is an example of how to use the NWS features in a console application:

```
#Create the NWS client interface
NWSMetricsClient *nwsClient = new NWSMetricClient(`myhost.com`,9000);

#Create the client to use the NWS client
DCNWSscheduler *scheduler = new DCNWSscheduler(nwsClient);

#Create an instance to use the DCNWSscheduler
DCInstance* instance;
    if ((status = DC->NewInstance(instance, layout, scheduler)) != DC_ERR_OK) {
        dcErrExit("failed GetFilterInstance: %s", DC_strerror(status));
    }
```

You can find out more about NWS at <http://nws.cs.ucsb.edu/>