DataCutter User's Guide
Version 2.1

Michael Beynon[†] Umit Catalyurek[‡] Christian Hansen[†]
Tahsin Kurc[‡] Joel Saltz[‡] Alan Sussman[†]

[†] Dept. of Computer Science
University of Maryland
College Park, MD 20742
{beynon,chansen,als}@cs.umd.edu

[‡] Dept. of Biomedical Informatics
Ohio State University
Columbus, OH 43210
{catalyurek.1,kurc.1,saltz.3}@osu.edu

September 23, 2002

1 Introduction

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 dataflow 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.

This documentation is intended to describe the prototype implementation in sufficient detail to enable the reader to configure the DataCutter framework and develop a new application.

2 Installation

This section presumes that a copy of the DataCutter source is present on the system on which it will be installed. DataCutter may be obtained as gzipped tar file from DataCutter project page found at http://www.cs.umd.edu/projects/hpsl/chaos/ResearchAreas/dc/.

DataCutter has been compiled and tested on several platforms. Table 1 lists these platforms.

Platforms
Linux kernel 2.2 or 2.4, running on x86 processors
Sun Solaris 2.x or 8, running on SPARC processors
IBM AIX 4.3.3 or greater, running on RS6000 processors

Table 1: Available platforms

To compile, the following environment variables may need to be set for the system in the case when they cannot be determined automatically during the build process:

- `CXX` - which C++ compiler to use (e.g., for AIX, `xlC_r` is the correct compiler)
- `PTHREAD_CXXFLAGS` - compiler args for using pthreads (e.g., `-D_REENTRANT`)
- `PTHREAD_LIBS` - linker args for correct pthread libs (e.g., `-lpthread`)

Compilation from gzipped tar files proceeds as follows:

```
> gunzip DataCutter-2.1.tar.gz | tar -xvf -
> cd DataCutter-2.1
> ./configure
> gmake
> gmake install
```

The configure step accepts various command line options. Several important examples are given here, however, for a full listing use `./configure --help`:

```
--enable-dird-host=<hostname>   [default=$HOST]
--enable-dird-port=<port>       [default=6100]
```

Once built, be sure that DataCutter-2.1/bin is in $PATH.

3 Quick Start

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

The first entity that must be running, is the directory daemon, which provides a centralized repository of contact information. A provided utility, `boot_util`, is used to start the directory daemon as follows:

```
> boot_util dird start
```

By default, an Xterm is created, and dird is started on the configure specified host:port. If this step fails, it may be that dird was not found in your $PATH. Next, we need application daemons running on any host we may want to use for placement of application filters. This is accomplished as follows:

```
> boot_util appd start condor.cs.umd.edu candida.cs.umd.edu
```

This will also cause an Xterm to be created by default for each host running appd. Note that parallel machines with schedulers require care to start the required processes. See `DataCutter-2.1/run/startappd.pbs` for an example that works with the PBS scheduler common on Linux clusters. At this point, the supporting services (dird and appd) are in place, and we can execute a DataCutter application. We choose the `example-detach` application for illustration:

```
> ln -s ../DataCutter-2.1 ~/dc
> cd ~/dc/example
> ./example-detach 1 10 1
```

This process becomes the ”console process”, and using the DataCutter API, creates instances of filters on the hosts running appd and sends work to them for processing. Output from the application then provides startup and status information similar to the following:

```
DataCutter Filter Service library v2.14 (01/10/2002)
pid=9294, listening on schisto.cs.umd.edu:49649
example-detach: [placement]
P=condor.cs.umd.edu
```

```
F=condor.cs.umd.edu 2
F=candida.cs.umd.edu
C=candida.cs.umd.edu 2

example-detach: 1 batches of 10 uow per batch, each appended to 1 instances
app: wh=0 finished, ret=0
app: wh=1 finished, ret=0
app: wh=2 finished, ret=0
      :
```

Note: If two unique hosts are unavailable, try substituting the application `test-minimal` (no arguments) for `example-detach`.

4 Configuration

Applications can specify the layout (what filters the application will use, and how they are connected) and the placement (mapping of filter to hostname) two ways: (1) in the C++ code by calling API functions, or (2) in separate configuration files. There is also a configuration file for the console process which determines the application's overall behavior. This section provides a description of the format and contents of the three types of configuration files used by DataCutter.

4.1 Console

A console configuration file is required for an application to run. By default, all applications will use the system-wide file installed by default into `DataCutter-2.1/etc/console.cfg`. However, application specific configurations will be used if a file named `<appname>.cfg` can be found in the current working directory. A console configuration file begins with the line `[console]` and contains the following variables:

dird_host	Connect to `dird` on this host
dird_port	Connect to `dird` on this port
sink_queue_size	Set the size of the sink queue
work_queue_size	Set the size of the work queue
stream_write_policy	Set the default write policy for streams
debug_src_dir	Turn on/off remote debugging
debug_extra_cmd	Commands to send to the debugger on startup
filterexit_stats_file	Enable output of filter statistics on exit
remote_setenv	Set enviroment variables on remote hosts
remote_x11_display	Enable automatic handling of remote $DISPLAY

Table 2: Console configuration options

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.

4.1.1 Queue Sizes

The variables sink_queue_size and work_queue_size determine the size of the remote process filter queues.

```
[console]
# sizes of remote process filter queues
sink_queue_size = 20;
work_queue_size = 20;
```

4.1.2 Stream Write Policies

The variable stream_write_policy 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

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

4.1.3 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 ”debug_src_dir” line to the `console.cfg` or `<app>\_console.cfg` 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 ”debug_extra_cmd” 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 filesystem, and readable from all remote processes, due to the creation of temporary files in the current run directory for the debugger command input script.

```
[console]
# enable remote process started in gdb/dbx
debug_src_dir = ../example ../src ../src/lib
debug_extra_cmd = 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.

4.1.4 X11 Remote Display

The appd daemon can be started in one of two modes: "xterm-ssh" or "daemon", determined by the "-noxterm" option to `boot_util`. 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 that 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 tunnelling 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 backend nodes, the cookie installed by Ssh on "front" needs to be transferred to the backend 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 filesystem is used across "front" and all the backend nodes. Given the non-negligible overhead in adding the remote cookies in the required way, `boot_util` 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
# foreward console process $DISPLAY setting (default = true)
set_x11_display = true
```

To run in daemon mode:

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

5. `front>` (run applications)

To run in xterm node:

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

4.2 Layout

The layout configuration file specifies the stream connections between the various filers. Here is a sample file of this type, `layout.cfg`:

```
[layout]
[filter.P]
outs = pf
[filter.F]
ins = pf
outs = fc
[filter.C]
ins = fc
```

4.3 Placement

The placement configuration file specifies on which hosts the filters will run.

Note: when specifying placement either in a config file, or using the C++ API, be sure not to have multiple entries for the same ¡filter¿ and ¡host¿ pair. If more than one type of filer is needed on a host, use the copy syntax of the placement config file to indicate this.

Incorrect:

```
[placement]
f1 = host1
f1 = host1
f2 = host2
```

Correct:

```
[placement]
f1 = host1 2
f2 = host2
```

4.3.1 Enhanced Syntax

Consider the following filter placement with the layout as given in Section 4.2:

placement.cfg:

```
[placement]
P = host1 1
F = host1 1
P = host2 1
F = host2 1
P = host3 1
F = host3 1
C = host4 1
```

There are three copysets of filter F located on host1, host2 and host3. We may want to restrict the scope of which sink copysets are considered for writing. For example, we may want filter P on host1 to only send to the filter F copyset colocated on host1. To specify this, the ambiguity between the P's and F's needs to be resolved. The following enhanced syntax for placement cfg files addresses this need.

```
[placement]
filter-name[:copy-set-id] = host [min-copies [max-copies]]
     :
[stream-name = write source-re sink-re { <none> | policy [policy-args...]] }
     :
```

Filter names can be appended with an optional copy-set-id to distinguish between multiple copysets. In addition, we can optionally add stream-name rules containing regular expressions for filter source and sink names, and specify the exact write policy to be used between them. Rules are examined in the order they appear until the *first* match for a filter (source, sink) pair. If no rules are present, there is an implied rule between all copysets using the default stream write policy specified in console.cfg, so existing placement.cfg files should work as before without modification. Note that a user-defined (UD) policy specified in the filter layout [stream.*] section, will override any of these rules or the console.cfg default policy. Using "¡none¿" for a policy stops the matching process to avoid matching with later rules that we do not want, and is shown in the first example below.

Restriction: Any single copyset can only use one write policy for sending, because it is undefined where a buffer should be sent if there is more than one write policy. By definition, a single write policy is used at a source to choose a copyset sink for sending. For writing, if a source filter was allowed to have one sink copyset that uses say RR and another sink copyset uses DD, then it is unclear how these two policies should be mixed to choose a destination. Therefore, it is disallowed, and rules that cause this situation will produce a warning message. For reading, there is not a similar problem, and the policies can be mixed on a source copyset basis. An example of this is a copyset on one host using a policy different from that of another copyset on another host, and both sending to a single sink copyset. For the above P-F-C example, assume the output of P on host1 and host2 is to be limited to the colocated copysets of F on host1 and host2 respectively, and the third copyset of P on host3 is free to write to any F copyset. A placement.cfg that supports this mode of operation could be written as:

```
[placement]
P:1 = host1 1
F:1 = host1 1
P:2 = host2 1
F:2 = host2 1
P = host3 1
F = host3 1
C = host4 1

pf = write P:1 F:1 rr
pf = write P:2 F:2 rr
pf = write P:1|P:2 * <none>
pf = write * * dd
fc = write * * rr
```

The round robin (RR) policy is used, since there is no advantage to the ACKs, since every source filter has only one sink. The P copies on host1 will now only write to the colocated F copies, and the same is true on host2. All other copies of P will write to all other copies of F using DD. The ”¡none¿” entry is used to avoid cases like P:1 or P:2 writing to the F filters on host3 using the DD policy.

This could have also been accomplished using the following alternative specification, but without using the ”¡none¿” entry. Here we used a copyset name for *each* P copyset, and explicitly give the policy for each. This method works well if there are not many other P copysets. If there are many other P copysets, the previous solution is simpler.

```
[placement]
P:1 = host1 1
F:1 = host1 1
P:2 = host2 1
F:2 = host2 1
P:3 = host3 1
F = host3 1
C = host4 1

pf = write P:1 F:1 rr
pf = write P:2 F:2 rr
pf = write P:3 * dd
fc = write * * rr
```

5 Programming Tutorial

This section provides an introduction to the DataCutter programming API which allows users to define custom filters and applications. Central ideas to the coding and compilation process will be presented in the context of an annotated example (`example-sort.cpp`). This simple application sorts sets of random numbers using three filters: a ”Producer” which generates the random numbers, a ”Sorter” which calls the C library's quicksort function on these sets, and a ”Consumer” which outputs the sorted sets to a file specified by the user. This example only covers part of the functionality of the DataCutter library. See the class header files for a more complete description.

5.1 Include Files

All of the following class and function definitions used are contained in the single header file `DataCutter.h` which must be referenced with:

```
#include <DataCutter.h>
```

5.2 Error Reporting

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

```
DC_strerror(int)
```

5.3 Filter Creation

Filters are created by subclassing the abstract base class `DC_Filter_Base_t` 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, calls to the `init`, `process`, and `finalize` functions are made in succession. 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 size objects per call.

5.3.1 Buffers and Work

The DC_Buffer_t 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 a 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, the consume method should be called, which may free the space. Heap allocated memory is handled correctly based on the constructor used. Generally avoiding static DC_Buffer_t objects with the use of consume is advised, unless setConsume(true) is explicitly called.

Activity	Example

Allocate memory	`const int setElements = 10;`
	`int bufSize;`
	`DC_Buffer_t buf;`
	
	`bufSize = setElements * sizeof(int);`
	`buf.New(bufSize);`

Set the amount of the buffer to be used	`buf.setSize(bufSize);`

Get a pointer to the beginning of the buffer	`int *p;`
	
	`p = (int*)buf.getPtr();`

Get the size of the current buffer	`for (unsigned int i = 0; i < buf.getSize(); ++i) {`
	`    :`
	`}`

Clear the buffer	`buf.Empty();`
Destroy the buffer	`buf.consume();`

Table 3: Common tasks using buffers

The DC_Work_t is the definition of a particular processing job or query. It is a container class for data objects DC_Buffer_t buf, which allows the parametrization 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 infomation is provided to each filter on initialization.

5.3.2 Initialization

The init 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 init(initarg_t&)
```

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

```
DC_RTN_t init(initarg &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 occured 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 (see Section 5.4).

For the `example-sort.cpp` example, a unit of work is defined in terms of the file name into which sorted numbers are written. Thus only the Consumer class is concerned with the contents of this buffer, which it uses for file creation. 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 Producer class must create a buffer in which it can store generated random numbers:

```
const int setElements = 10;

class Producer: public DC_Filter_Base_t {
  int bufSize;
  DC_Buffer_t buf;

  DC_RTN_t init(initarg_t &initarg) {
    bufSize = setElements * sizeof(int);
    buf.New(bufSize);
    buf.setSize(bufSize);
    return DC_RTN_OK;
  };
      :
```

As the Sorter class has no special memory requirement and can use the buffer provided on reading from its input stream, its `init` function is simply:

```
class Sorter: public DC_Filter_Base_t {
public:
  DC_RTN_t init(initarg_t &initarg) {
    return DC_RTN_OK;
  };
      :
```

For the Consumer class, an output file stream must created using the name passed through the work object:

```
class Consumer: public DC_Filter_Base_t {
public:
  ofstream ofs;

  DC_RTN_t init(initarg_t &initarg) {
    char* outfile = initarg.pwork->buf.getPtr();
    ofs.open(outfile);
    return DC_RTN_OK;
  };
      :
```

5.3.3 Processing

Once init 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(arg_t&)
```

The argument of type arg_t is derived from initarg_t 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 colocated 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 colocated with the sink this will be deposited directly in that queue. Use of stack allocated DC_Buffer_t 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 colocated consumer.

In example-sort.cpp the Producer's task during the process stage is to generate sets of random numbers and send them to the Sorter filter:

```
DC_RTN_t process(arg_t &arg) {
```

Activity	Example
Get an input stream index by name	`int inIndex = arg.insIndex("P-S");`
Get an output stream index by name	`int outIndex = arg.outsIndex("S-C");`

Read from the input stream and write to the output stream	DC_Buffer_t *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();
	}

Complete processing	`return DC_RTN_OK;`

Table 4: Working with streams

```
  int *p;

  p = (int*)buf.getPtr();

  // Get the stream index
  int outIndex = arg.outsIndex("P-S");
  if (outIndex < 0) {
    cerr << arg.sbFilterName << ':' << DC_strerror(outIndex) << endl;
    return DC_RTN_ABRT;
  }

  // Create random number sets and write them to the stream
  srand((unsigned)p);
  for (int j = 0; j < numSets; ++j) {
    for (int i = 0; i < setElements; ++i) {
      p[i] = rand() % 100;
    }
    if (arg.outs[outIndex].write(&buf) != DC_ERR_OK) {
      cerr << arg.sbFilterName << ": failed write to buffer " << buf << endl;
    }
  }

  // Indicate that this work cycle has ended
  return DC_RTN_OK;
};
```

For the Sorter class, its process stage reads in these sets one at a time, uses the C library's qsort routine to handle the actual "work", and then forwards the result to the Consumer filter:

```
DC_RTN_t process(arg_t &arg) {
  DC_Buffer_t *pbuf;

  // Get the input stream index
  int inIndex = arg.insIndex("P-S");
  if (inIndex < 0) {
    cerr << arg.sbFilterName << ':' << DC_strerror(inIndex) << endl;
    return DC_RTN_ABRT;
  }

  // Get the output stream index
  int outIndex = arg.outsIndex("S-C");
  if (outIndex < 0) {
    cerr << arg.sbFilterName << ':' << DC_strerror(outIndex) << endl;
    return DC_RTN_ABRT;
  }

  // Read, sort, and write out the number sets
  while ((pbuf = arg.ins[inIndex].read())) {
    qsort(pbuf->getPtr(), pbuf->getSize() / sizeof(int), sizeof(int), intcompare);
    if (arg.outs[outIndex].write(pbuf) != DC_ERR_OK) {
      cout << arg.sbFilterName << ": failed write on buffer " << pbuf << endl;
    }
    pbuf->consume();
  }
  return DC_RTN_OK;
};
```

Finally, the Consumer class during this stage must simply write out the received sets to a file opened during initialization:

```
DC_RTN_t process(arg_t &arg) {
  DC_Buffer_t *pbuf;

  // Get the input stream index
  int inIndex = arg.insIndex("S-C");
  if (inIndex < 0) {
    cerr << arg.sbFilterName << ':' << DC_strerror(inIndex) << endl;
    return DC_RTN_ABRT;
  }

  // Read and output the sorted sets to a file
```

```
    while ((pbuf = arg.ins[inIndex].read())) {
      int *p = (int*)pbuf->getPtr();
      for (unsigned int i = 0; i < pbuf->getSize() / sizeof(int); ++i) {
        ofs << p[i] << ' ';
      }
      ofs << endl;
      pbuf->consume();
    }

    return DC_RTN_OK;
  };
```

5.3.4 Finalization

After the process function has returned, the finalize 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 finalize(void)
```

For both the Producer and Sorter filters in example-sort.cpp, there is nothing to be done during this stage, so the function just returns:

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

In the case of the Consumer filter, the filehandle created during initialization needs to be closed:

```
DC_RTN_t finalize(void) {
  ofs.close();
  return DC_RTN_OK;
};
```

5.4 Application Creation

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 provided during system initialization. The prototype for this function is:

```
DC_Filter_Base_t *filter_factory(char *)
```

The run time system is initialized by creating an instance of the `DC_FilterService_t` class and calling its `init` function. Since 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 `isRemoteProcess`, and if such is the case, the process prepares itself to run filter code by calling `RemoteProcess`.

Activity	**Example**

Instantiate and initialize the run time system	`DC_FilterService_t DC;`
	
	`if (DC.init("example-ppfc", &filter_factory, &argc, &argv)) {`
	`  exit(1);`
	`}`

Check whether the current process is remote, and if so, hand over control to the run time system	`if (DC.isRemoteProcess()) {`
	`  DC.RemoteProcess();`
	`  return 0;`
	`}`

Table 5: Tasks involved in bringing up the DataCutter run time system

For the `example-sort.cpp`, three filters are defined, Producer, Sorter, and Consumer, which will later be refered to as ”P”, ”S”, and ”C” in the application configuration. For the run-time system to be able to generate these filters when needed, the following function is defined:

```
DC_Filter_Base_t *filter_factory(char *sbFilterName) {
  if (strcmp(sbFilterName, "P") == 0) {
    return new Producer;
  } else if (strcmp(sbFilterName, "S") == 0) {
    return new Sorter;
  } else if (strcmp(sbFilterName, "C") == 0) {
    return new Consumer;
  } else {
    cerr << "app: unknown filter \"" << sbFilterName << "\" to
create" << endl;
  }
  return NULL;
}
```

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

```
int main(int argc, char* argv[]) {
```

```
DC_FilterService_t DC;
if (DC.init("example-sort", &filter_factory, &argc, &argv)) {
  exit(1);
}

// Check whether this process is a remote copy for running filter instances
if (DC.isRemoteProcess()) {
  // Run any filter instances as directed by console process
  DC.RemoteProcess();
  return 0;
}
```

The task of the console process is to then:

1. Instantiate an object of type DC_Work_t 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 DC_Work_t and an object is created by:

```
DC_Work_t work;
```

An object of type DC_FilterLayout_t contains the member function Add which allows one to specify the logical connections between filters and streams. These are decided by giving the filter name, the input stream name, and the output stream name. When there is no input or output for a particular filter, NULL can be given as the stream name. Multiple inputs or outputs can be specified with a space separated list of filters. The program example-sort.cpp uses a single connection from Producer to Sorter and from Sorter to Consumer, so the layout is as follows:

```
DC_FilterLayout_t layout;
layout.Add("P", NULL, "P-S");
layout.Add("S", "P-S", "S-C");
layout.Add("C", "S-C", NULL);
```

Alternatively, it is also possible to read this information from a configuration file in the format specified in Section 4.2 with:

```
layout.ReadFile("layout.cfg");
```

Filter placement is handled using a `DC_Placement_t` with calls to its `Add` function, providing it the filter name and the hostname. The string ”¡one_per_node¿” allows for a filter to be placed on any one of the nodes running appd. This placement was chosen for `example-sort.cpp`:

```
DC_Placement_t placement;
if (DC.PlacementPlanning(layout, work, placement) != 0) {
  // Choose our own placement instead
  placement.Add("P", "<one_per_node>");
  placement.Add("S", "<one_per_node>");
  placement.Add("C", "<one_per_node>");
}
```

Activity	**Example**

Create an instance of a filter group	`DC_FilterInstance_t instance;`
	
	`if (DC.NewFilterInstance(layout,`
	`                         work,`
	`                         placement,`
	`                         instance) != DC_ERR_OK) {`
	`  cerr << "failed GetFilterInstance";`
	`}`

Prepare a work description	`work.buf.Empty();`
	`work.buf.Append(argv[i], strlen(argv[i]) + 1);`

Assign work to a filter	`int wh;`
	
	`if ((wh = instance.AppendWork(work)) < 0) {`
	`  cerr << "failed AppendWork: " << DC_strerror(wh) << endl;`
	`}`

Wait for the filter to finish processing the assignment	`int sts;`
	
	`if ((sts = DC.WaitWork(wh)) != DC_ERR_OK) {`
	`  cerr << "failed WaitWork: " << DC_strerror(sts) << endl;`
	`}`

Wait for the filter to finish processing any work assignment	`while ((wh = DC.WaitAnyWork()) != DC_ERR_NoWorkFound) {`
	`  if (wh < 0) {`
	`    cerr << "failed WaitAnyWork: " << DC_strerror(wh) << endl;`
	`  }`
	`}`

Terminate the instance	`if ((sts = DC.StopFilterInstance(instance)) != DC_ERR_OK) {`
	`  cerr << "failed StopFilterInstance: " << DC_strerror(sts) << endl;`
	`}`

Table 6: Managing a filter group

Having specified the layout and optionally, placement, it is now possible to create an instance of the filter group and assign it work. An instance is of type DC_FilterInstance_t and it is created with a call to the NewFilterInstance function of the DC_FilterService_t. 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 DC_FilterInstance_t object.

Before assigning additional work to one or more filters, the application can wait for the completion of the 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 unambigious. 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 StopFilterInstance.

The application example-sort.cpp uses only a single filter instance and assigns work to this instance work based on the file names provided at the command line.

```
// Get a filter instance (existing or new) for executing work
int status;
DC_FilterInstance_t* instance;

if ((status = DC.NewFilterInstance(layout, work, placement, instance)) !=
DC_ERR_OK) {
  cerr << "failed GetFilterInstance: " << DC_strerror(status) << endl;
}

// Append work to instance
int wh;

for (int i = 1; i < argc; ++i) {
  work.buf.Empty();
  work.buf.Append(argv[i], strlen(argv[i]) + 1);

  if ((wh = instance->AppendWork(work)) < 0) {
    cerr << "failed AppendWork: " << DC_strerror(wh) << endl;
  }
  status = DC.WaitWork(wh);
  if (status == DC_ERR_InstanceFail) {
    cerr << "failed WaitWork: " << DC_strerror(status) << endl;
    break;
  }
  cout << "finsihed writing to " << argv[i] << endl;
```

```
    }
    if ((status = DC.StopFilterInstance(instance)) != DC_ERR_OK) {
      cerr << "failed StopFilterInstance: " << DC_strerror(status) << endl;
    }
```

5.5 Compilation

Compilation requires use of the DataCutter library `libDC.a`. The following set of commands will compile `example-sort.cpp` found in the `dc/example` directory for Linux (other systems may have additional library requirements):

```
    g++ -DHAVE_CONFIG_H -I../include -DDEBUGGER_USE_GDB  -DLINUX -c -o example-sort.o example-sort.cpp
    g++ -DDEBUGGER_USE_GDB -DLINUX -L../lib -o example-sort example-sort.o -lDC -lnsl -lpthread -lm
```

A Java Filters

The definition of filters in the Java programming language is possible if Java support has been compiled in (see the INSTALL file). Java filters are defined in a fashion similar to C++, by subclassing `DC_Filter_Base_t`, found in the `dc.jfilters` class package, and implementing the three callback functions. However, due to a naming conflict these functions have been renamed `start_work`, `process`, and `finish_work`. An outline of a filter definition in Java follows:

```
import dc.jfilters.*;

public class DC_Filter_Test extends DC_Filter_Base_t {
    public DC_Filter_Test() {
        super();
    }

    public int start_work(initarg_t arg) {
:
    }

    public int process(arg_t arg) {
:
    }

    public int finish_work() {
:
    }
}
```

It is possible to use Java filters alone or in combination with C++ filters. In both cases, instantiation of Java filters requires the use of a modified `filter_factory` function, `DC_CToJava_Filter_t::filter_factory`, which creates the Java Virtual Machine needed to access the filter functions. The class `DC_CToJava_Filter` is specified in the header file `dc_ctojava_filter_t.h`.

B Environment Variables

Two variables, DATACUTTER_DIRDHOST and DATACUTTER_DIRDPORT, will override the compiled in defaults for the host and port that the dird will operate.

C Command Reference

Please refer to the man pages for further information on each of the commands described in this section.

C.1 Dird

Dird is the centralized directory used for locating appds.

Configuration

`dir.db`	Listing of running appds that allows reconnection when dird fails

Flags

`-port <port>`	Override the compiled-in default for the dird port
`-init-ack <host:port>`	Request acknowledgement of a successful starup
`-restore-state`	Read in `dir.db` to re-establish communication with running appds

C.2 Appd

Appd is the daemon which spawns user applications on remote hosts.

Configuration

`appd.cfg`	Specify pattern matching rules for executable name translation on various hosts

Flags

`-daemon`	Run as a daemon and send output to $TMPDIR or `/tmp`
`-outputhandler <host:port>`	Use a socket for output instead of stdout
`-host <host>`	Override host name if gethostname doesn't work
`-exit`	Exit appd after running a single filter application
`-nokillchild`	Don't kill appd child processes on exit
`-dird-host <host>`	Override compiled default for the dird host
`-dird-port <port>`	Override compiled default for the dird port
`-init-ack <host:port>`	Request acknowledgement of a successful startup