Active Storage User's Manual

October 2007

Juan Piernas, Jarek Nieplocha
Pacific Northwest National Laboratory

Index

This document describes the Active Storage framework from the user's point of view. It uses Lustre 1.6 as the parallel file system, although all the same is applicable to PVFS2. The current implementation also allows us to use a single computer with a traditional file system to run Active Storage jobs for debugging purposes. More information at http://hpc.pnl.gov/projects/active-storage.

1. Introduction

Active Storage assumes that, in a cluster with thousands of compute nodes and hundreds of OST's (the storage nodes of Lustre), both compute nodes and OST's are clients of the parallel file system, i.e., they mount the parallel file system on a given directory, for example, `/lustre`. Figure 1 shows the architecture of Active Storage.

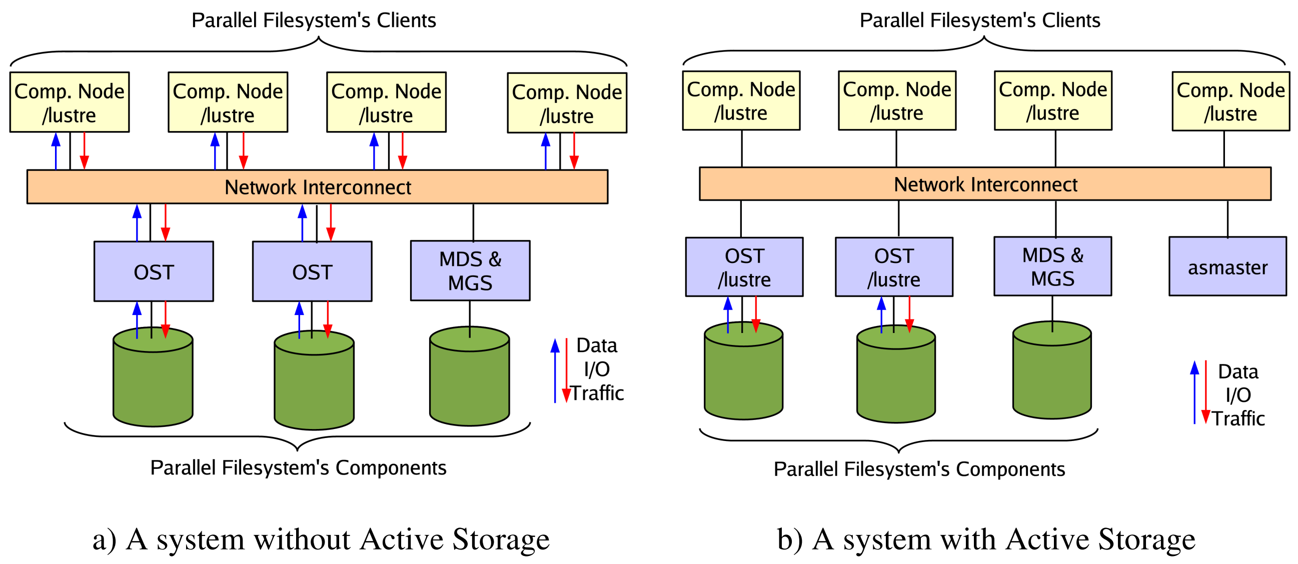


Figure 1: This figure shows the differences between a system without Active Storage (a), where the compute node process the data in the storage node causing a high network traffic, and a system with Active Storage, where every storage node processes the data which it stores.

One of the clients of the file system (either a compute node, a storage node, or even a dedicated node as in Figure 3.(b)) will run `asmaster`, a Python program which receives a *rule* describing an Active Storage job, and performs the actions contained in it (see below). In the storage nodes, there also exists an Active Storage Runtime Framework (ASRF), a set of programs which assist `asmaster` in executing a job.

A rule (a job) in the Active Storage framework specifies a program to run on every storage node as a *processing component*. The processing components are remotely run by `asmaster`, as we can see in Figure 2.

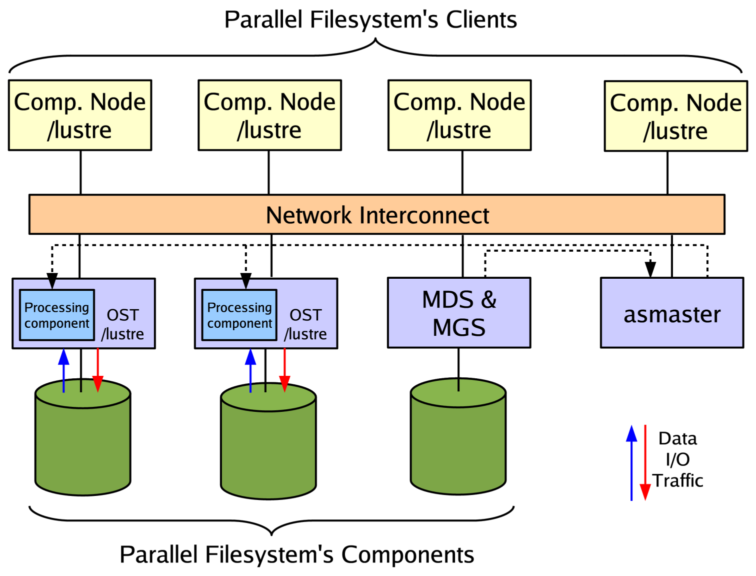


Figure 2: asmaster receives striping information from the meta-data server and remotely execute the processing components by using the rsh application.

The programs and rules used by Active Storage must be globally accessible. Since all the nodes use Lustre, it is a good idea to store the files in the parallel file system itself (although NFS, AFS or any other network file system are also possible options). Let us assume that there exists a directory called `/lustre/asd` which contains all the programs and libraries of Active Storage. In the future, this directory could have several subdirectories (`etc` for configuration files, `bin` for program binaries, `lib` for library files, etc.), but it contains a `lib` directory with the Active Storage libraries and an empty `bin` directory right now.

2. Rules

An Active Storage job is described by a rule. A rule is a simple XML file which, at least, has two elements: a file pattern specifying the files to be processed, and a path to the program to run on the storage nodes. The following is an example of a minimal (although maybe not very useful) rule:

```
<?xml version="1.0"?>
<rule>
    <match>
        <pattern>/lustre/molecule*</pattern>
    </match>
    <program>
        <path arch="any">/bin/ls</path>
    </program>
</rule>
```

This rule can be executed as:

```
      /lustre/asd/asmaster /lustre/rule.xml
```

Note that the rule file is globally accessible via the parallel file system. This is important because the Active Storage Runtime Framework in every storage node must read the contents of the rule before running the corresponding processing component.

The following examples illustrate how a rule works. By now, we assume that the input files are not striped, i.e., every input files is stored in only one node. The treatment of striped files is described in section 4.

Example 1:

```
<?xml version="1.0"?>
<rule>
    <match>
        <pattern>/lustre/molecule*.in</pattern>
        <from>*.in</from>
        <to>*</to>
    </match>
    <program>
        <path arch="any">/lustre/bin/weight</path>
        <arguments>@.in @.out</arguments>
    </program>
</rule>
```

According to this rule, for every file which matches the pattern, asmaster will run a process with the given program, `/lustre/bin/weight`, on the corresponding storage node. The process will receive two arguments: an input file (@.in) , and an output file (@.out). Actually, @ is a special character which represents a string. This string is the name of the matching file or part of it, and its value depends of the `<from>` and `<to>` elements of a rule. For example, if the matching file is `/lustre/molecule1.in`, then the @ symbol will be equivalent to the string `/lustre/molecule1`, because `<from>` says that the string ".in" must be removed from the name and `<to>` does not add any new character. Therefore, the first argument of the process will really be `/lustre/molecule1.in` and the second one will be `/lustre/molecule1.out`.

In our example, the @ character has another meaning. It means that the output file must be created (if it does not exist yet) in the same storage node as the input file. In this way, I/O operations can be completely local. There are exceptions to this rule. See **@ attributes** below.

If another file, `/lustre/molecule23.in`, exists in a different OST, Active Storage will do the same: it will create the file `/lustre/molecule23.out`, and run the program `weight` on that OST, passing both file names as arguments. The resulting system can be seen in Figure 3.

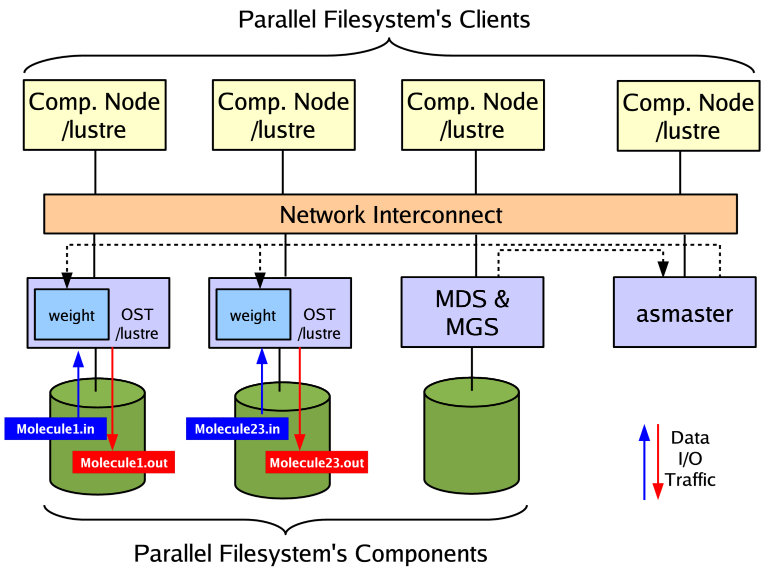


Figure 3: Active Storage run a processing component per matching file and create output files in the same storage node as the corresponding input file. In this way, I/O operations are local

If there are thousands of files which match the pattern, Active Storage will create thousands of processing components, one per file, at the same time. If two or more files, whose names match the pattern, exist in the same storage node, Active Storage will run several instances of the same program in the same OST. Every processing component will run locally and independently on the storage node. We can limit the number of processes per node by assigning a value to the `<multiproc>` element. For example, if we want to run only one process at a time, the rule must have the following:

```
<match>
    <pattern>/lustre/molecule*.in</pattern>
    <from>*.in</from>
    <to>*</to>
    <multiproc>1</multiproc>
</match>
```

With the above rule, `asmaster` will run a processing component per matching file, wait for the completion of the different processes and then exit. If the application running on the compute node is to create new files, and we want `asmaster` to wait for more matching files, we have to explicitly say that by means of the `<trigger>` element:

```
<match>
    <pattern>/lustre/molecule*.in</pattern>
    <from>*.in</from>
    <to>*</to>
    <multiproc>1</multiproc>
    <trigger>yes</trigger>
</match>
```

Example 2:

If the program to run is able to read from the standard input, and write to the standard output, then the rule in the previous example can be written as:

```
<?xml version="1.0"?>
<rule>
    <stdfiles>
        <stdin>@.in</stdin>
        <stdout>@.out</stdout>
    </stdfiles>
    <match>
        <pattern>/lustre/molecule*.in</pattern>
        <from>*.in</from>
    </match>
    <program>
        <owner>piernas.piernas</owner>
        <path arch="any">/lustre/bin/weight</path>
    </program>
</rule>
```

Since the `<stderr>` element does not appear, the standard error output will be redirected to the `/dev/null` device (this is the default behavior for the standard input and output, unless otherwise specified as in this example). Note that the `<to>` element has been deleted (its default value is already “*”), and that the `<owner>` element has been added to specify the user and group which the processing components will belong to. If it does not appear, the processing components will belong to the same user and group as the `asmaster` process's.

Also note that, with the redirection mechanism, even typical Unix commands such as `gzip`, `wc`, `grep`, `sed`, `awk`, etc., can be easily used without any modification.

<u>Example 3:</u>

```
<?xml version="1.0"?>
<rule>
    <match>
        <pattern>/lustre/molecule*.in</pattern>
        <from>*.in</from>
    </match>
    <program>
        <path arch="any">/lustre/bin/weight</path>
        <arguments>@{wait}.in @.out</arguments>
    </program>
</rule>
```

This rule is similar to that given in the first example, but with an important difference: the first argument to pass to the program is "@{wait}.in", not "@.in". "wait" is an attribute of the file specified by "@.in", and means that a processing component will not see the end of the input file until the remote client which writes to it explicitly specifies that it has closed the file. That is, "wait" says that a processing component will wait for more data after reaching the end of the file, unless the remote client has closed the file. This mechanism is transparent to the processing components, and works as long as they use the `read` or `fread` function to read from the input file (functions such as `pread` or `readv` are not supported yet, although no many programs use them).

In the current implementation, a file close is signaled by the remote client by creating an empty file with the same name plus the `.endas` suffix (for example, `/lustre/molecule1.in.endas`)[1].

<u>Example 4</u>:

```
<?xml version="1.0"?>
<rule>
    <match>
        <pattern>/lustre/molecule*.in</pattern>
    </match>
    <program>
        <path arch="any">/lustre/bin/weight</path>
        <arguments>@{wait} @.out @.composition</arguments>
    </program>
</rule>
```

1 In Lustre 1.6, there is no way to know when a remote client has closed a file, so we need this mechanism or something similar.

This rule works like the previous one, but note that there are not `from` and `to` elements. That means that @ is equal to the input file, so the first argument to be passed to the program will be `/lustre/molecule1.in` and `/lustre/molecule1.in.out` the second one. Active Storage will also create another file, `/lustre/molecule1.in.composition`, in the same OST. This file will be passed as the third argument to the program `weight`. Note that it is easy to support 1IN → XOUT patterns (1 input file, X output files).

Example 5:

```
<?xml version="1.0"?>
<rule>
    <match>
        <pattern>/lustre/molecule*.in</pattern>
    </match>
    <program>
        <path arch="any">/lustre/bin/weight</path>
        <arguments>@{wait} @.out @.composition</arguments>
    </program>
    <time>
        <start>2007-5-11 10:0:0</start>
        <end>2007-5-11 12:0:0</end>
    </time>
</rule>
```

Again, this rule works like the previous one, but it also specifies start and end times. If a processing component takes more than two hours (the difference between the start and end times) to complete, it will be killed (which is the default behavior unless the `<kill>` element says another thing).

To Summarize, the only two things that an user has to do is to write a program to be run by the "processing components" (or use one of the many filter commands which already exist in an Unix environment), and submit the corresponding rule to the Active Storage system.

3. Variable expansion

The file names which appear in the `<stdin>`, `<stdout>`, `<stderr>`, `<arguments>` and `<delete>` elements can contain environment variables names which will be expanded during the rule execution. The variables must be given as ${variable-name}. For example, `${USER}` will be replaced with the user's name. If a variable does not exist, it will be replaced with the `NOVAL` string (note that a shell like bash usually replaces a non-existing variable with an empty string).

Active Storage also defines the following seven variables per processing component:

- NODENAME: host name of the storage node.
- NODENUM: node index of the storage node.
- CHUNKOFFSET: offset of the first chunk stored in the storage node.
- CHUNKNUM: number of storage nodes used by the file.
- CHUNKSIZE: chunk size.
- ABSCHUNKNUM: contains a six-figure number which is the number of chunks the matching file is split into.
- ABSCHUNKOFFSET: this variable contains a six-figure number which is the absolute offset of the chunk to be processed in the input file, starting at 0.

The last environment variable, ABSCHUNKOFFSET, is defined only when the value of the `<perchunk>` is yes. The usefulness of these variables can be seen is in the following section.

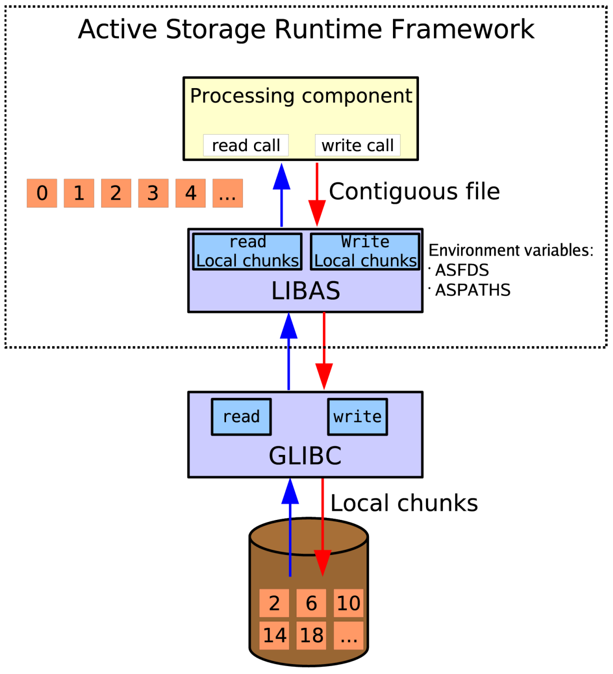
The expansion of environment variables takes place before the interpretation and expansion of the @ symbol.

4. Striped files

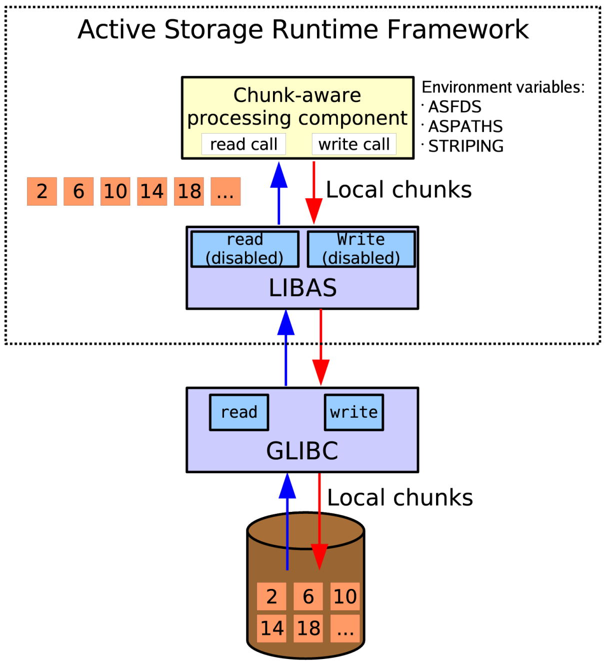
The management of striped files is a little complex. The first issue to address is the visibility of the file chunks on the processing components' side. If the file records[2] can be processed individually, and they are chunk-aligned, then the processing component in a storage node can see all the local chunks as a single, contiguous file, and the I/O will be entirely local.

However, if the file records are not aligned, or a processing component needs access to several records at the same time, then every processing component must see the whole file(s), but it must also receive information about which chunks are stored in its storage node. This is achieved via the environment variables CHUNKOFFSET, CHUNKSIZE, CHUNKNUM, NODENUM and STRIPING. In this case, the I/O operation will be "mostly" local, because some processing components will read record portions stored in other storage nodes. Note that is up to the processing components to perform I/O operations properly, i.e., Active Storage will provide them with striping information about the different files, but each processing component will have to use that information to decide which records to process. Figure 4 shows the differences between a transparent and non-transparent access to striped files.

2 In our context, the records are the data structures stored in the file.



(a) Transparent access to striped files



(b) Non-transparent access to striped files

Figure 4: In (a), the Active Storage Runtime Framework passes striping information to the libas library via several environment variables, ant the library intercepts I/O function calls to provide a transparent access to the striped files. In (b), the processing component is aware of the files' layouts, and also receive striping information via environment variables.

The variables CHUNKOFFSET, CHUNKSIZE and CHUNKNUM only have striping information about the matching file. However, if a processing component needs striping information about other files, then our rule must provides their names in a `getstripe` element, and Active Storage will pass that information to the processing components in the environment variable STRIPING. This variable is a colon-separated list of elements, where each element has the following syntax:

filename#chunkoffset#chunksize#chunkfirst#chunknum

Chunkoffset, chunksize, and chunknum have the same meaning as the corresponding environment variables, but with respect to the given file. Chunkfirst is the storage node which contains the first chunk of the file. Note that chunkoffset is usually equal or greater than 0, but it can be -1 if the storage node of the processing component does not contain any chunk of the file.

The visibility of the file chunks is managed by the “hidechunks” attribute of the @ symbol. By default, the processing components will see whole files. With hidechunks, a processing component will see all the local chunks as a single, contiguous file. This option can be applied to both input and output files.

The following rule shows the use of the hidechunk attribute:

Example 6:

```
<?xml version="1.0"?>
<rule>
    <match>
        <pattern>/lustre/doubles.in</pattern>
    </match>
    <program>
        <path arch="any">/lustre/dscal</path>
        <arguments>12345.67890 @{hidechunks}
            @{copystriping,hidechunks}.out</arguments>
    </program>
</rule>
```

Note that, in order to create the output file with the same striping as the input file, we have to use the copystriping attribute, because the default behavior is to create non-striped files.

In our current implementation, the hidechunks attribute is transparently supported for processing components which use the `read, fread, write` and `fwrite` system calls for file I/O.

Another problem when processing a striped file is that the output file, if any, can have a different record size and can contain a different number of records. Although both the input and the output files can have the same striping, an input record and its corresponding output record can be in different chunk offsets and, hence, in different storage nodes.

To make the output operations local, a solution is to create an output file per storage node, and to store in it all the output records produced by the processing of the input records in the local chunks. In this case, the name of the output file must use the NODENUM or NODENAME variables to create an unique output file per node. The following rule provides an example of this use:

Example 7:

```
<?xml version="1.0"?>
<rule>
```

```
    <match>
        <pattern>/lustre/doubles.in</pattern>
    </match>
    <program>
        <path arch="any">/lustre/dscal</path>
        <arguments>12345.67890 @{hidechunks}
            @.out-${NODENAME}</arguments>
    </program>
</rule>
```

With one output file per storage node, each output file will contain non-consecutive output records, because a storage node does not store consecutive chunks of a striped file. If we want to access the output records in the same order as the corresponding records in the input file, one option is to save in each output file information about which records it contains.

Another option is to create an output file per chunk (our implementation can do this for only the chunks of the matching file). If the file name contains the absolute offset of the input chunk, it will be very easy to access the records in the output files in the same order as its corresponding records in the input file, or to concatenate all the output files to create a single file with all the output records. The following rule shows this option:

Example 8:

```
<?xml version="1.0"?>
<rule>
    <match>
        <pattern>/lustre/doubles.in</pattern>
    </match>
    <program>
        <path arch="any">/lustre/dscal</path>
            <arguments>12345.67890 @{hidechunks}
                @.out-${ABSCHUNKOFFSET}</arguments>
        <perchunk>yes</perchunk>
    </program>
</rule>
```

Note that the value of the `<perchunk>` element is “yes” and that the third argument of the program (the output file in our case) uses the variable ABSCHUNKOFFSET, which is expanded to the absolute offset of the chunk being processed. After carrying out this rule, there will be a lot of output files, one per chunk, which can be concatenated into a single file by using, for example, the following command:

```
            cat /lustre/doubles.in.out-* > doubles.in.out
```

In it is interesting to note that we are also using the "hidechunk" attribute for the input file (second argument of the program). Therefore, all the processing components will transparently see and process only one chunk of the file.

Under certain circumstances, the `<perchunk>` option can produce unexpected results if the matching file grows during its processing. `asmaster` will show a warning message if it detects that the file size has changed during the Active Storage job.

It is important to remember that Active Storage will try to create any file with a @ symbol in its name (unless you specify the "nocreate" attribute). If the file already exists, it will remain unchanged (including its striping). It is also worth noting that, either the matching file is striped or not, every processing component always receives the environment variables CHUNKOFFSET, CHUNKSIZE, and CHUNKNUM.

5. The mapper component

Until now, we have assumed that all the data in the input files is processed. If Active Storage processing applies only to a part of the data, either contiguous or not, in the files, it becomes necessary to know which storage nodes have relevant data in order to run processing components only on them. To achieve this, it is possible to specify a *mapper* in a rule. The mapper is a program which receives, among other arguments, a file and its striping information, and return a list (a *map*) of the storage nodes which contain data to process. Without a mapper, all the storage nodes used by each input file run processing components.

The mapper is useful for processing files with complex data structures, such as netCDF or HDF5 files. The following examples show the use of the mapper with netCDF files:

Example 9:
```
<?xml version="1.0"?>
<rule>
    <stdfiles>
        <stdout>@.out-${NODENAME}</stdout>
    </stdfiles>
    <match>
        <pattern>/lustre/netcdf/data.in</pattern>
    </match>
    <program>
        <path arch="any">/lustre/netcdf/processdata.py</path>
        <arguments>@ ta</arguments>
    </program>
    <mapper>
        <path arch="any">/lustre/netcdf/netcdfmapper.py</path>
        <arguments>@ ta ${CHUNKNUM} ${CHUNKSIZE}</arguments>
    </mapper>
</rule>
```

In this rule, the mapper receives four arguments: the matching file, the name of the variable we want to process, the number of storage nodes used by the file, and the stripe size. The mapper then returns the list of storage nodes which have data of the variable. This list is a blank-separated string of positive integers, including 0. The first chunk of the file is always in storage node 0, and the last storage node is always N-1, where N is the number of storage nodes used by the file. That is, the numbers returned by the mapper are not the real names or ID's of the storage nodes in the parallel file system.

For example, if a files is striped across the OST's 3, 4, and 5 of a Lustre file system, and the mapper returns “0 2”, it means that the data of the variable is in the OST's 3 and 5 of the file system. In PVFS, this could means that the relevant data is in the storage nodes “node14” and “node15”.

Note that, in the selected storage nodes, the processing components must locate the data they have to process by using the striping information received from `asmaster`.

6. @ attributes

The @ symbol can have attached a list of options, or *attributes*, between curly brackets. The different attributes are:

- `wait`, `nowait`: whether the program must wait for the `.endas` file or not. Default: nowait.
- `hidechunks`, `nohidechunks`: whether the different chunks of a file stored in the same storage node must be seen as a single, contiguous file or not. Default: nohidechunks.
- `create`, `nocreate`: whether the file must be created or not. Default: create.
- `nostriping`, `defaultstriping`, `copystriping`, `setstriping`: when the file must be created, either if it must be stored in only one storage node, must be created with the default file system striping, must use the same striping as the matching file, or must be created with the given striping information. The syntax of the last option is `setstriping=stripe-size:start-node:stripe-cnt`, where `stripe-size` is the chunk size, `start-node` is the node which will contain the first chunk of the file, and `stripe-cnt` is the number of storage nodes to be used.

The interpretation and expansion of the @ symbol takes places after the expansion of the environment variables. Without attributes, the default values are used.

7. Use of a local file system

By default, Active Storage assumes that we are using a Lustre file system. But it is possible to use a single computer and a local file system to install Active Storage and run jobs. It is also possible to specify a "default striping" for all the files to test if the mapper and the processing components are working properly (obviously, this is a "virtual" striping because there is only one "storage node").

The following rule uses a local file system where all the files are spread across 8 "storage nodes" using a 1MB (1048576 bytes) "stripe size":

Example 10:

```
<?xml version="1.0"?>
<rule>
    <stdfiles>
        <stdout>@.out-${CHUNKOFFSET}</stdout>
    </stdfiles>
    <match>
        <pattern>/tmp/netcdf/data.in</pattern>
    </match>
    <program>
        <path arch="any">/tmp/netcdf/processdata.py</path>
        <arguments>@ ta</arguments>
    </program>
    <mapper>
        <path arch="any">/tmp/netcdf/netcdfmapper.py</path>
        <arguments>@ ta ${CHUNKNUM} ${CHUNKSIZE}</arguments>
    </mapper>
    <filesystem>
        <type>localfs</type>
        <striping>8:1048576</striping>
    </filesystem>
</rule>
```

Note that we use the CHUNKOFFSET variable and not NODENAME to create the output files because all the processing components run on the same node which only has one name. Therefore, we need other information to differentiate between processing components, and create an unique output file for each one.

8. Processing patterns

Basically, the processing components in the above examples use a 1IN → 2OUT or 1IN → 3OUT, because there is only one input file, and 2 or 3 output files. The following examples illustrate other processing patterns:

Example 11:

```
<?xml version="1.0"?>
<rule>
    <match>
        <pattern>/lustre/molecule*.in</pattern>
        <from>*.in</from>
    </match>
    <program>
        <path arch="any">/lustre/bin/weight</path>
        <arguments>@.in @.out</arguments>
        <delete>@.in</delete>
    </program>
</rule>
```

Note that the input file will be deleted after the process exits.

Example 12:

```
<?xml version="1.0"?>
<rule>
    <match>
        <pattern>/lustre/molecule*.in</pattern>
    </match>
    <program>
        <path arch="any">/lustre/bin/weight</path>
        <arguments>@</arguments>
        <delete>@</delete>
    </program>
</rule>
```

There is not output file, and the input file is deleted after the program execution.

Example 13:

```
<?xml version="1.0"?>
<rule>
    <match>
        <pattern>/lustre/molecule*.in</pattern>
        <from>*.in</from>
    </match>
    <program>
        <path arch="any">/lustre/bin/weight</path>
        <arguments>@.in @.out1 @.out2 @.out3 .....</arguments>
    </program>
</rule>
```

There are one input file and several output files in the same OST, which are passed as arguments.

9. Programming and performance tips

The layout (striping) of the input files is probably determined by the I/O requirements of the application which is run on the compute nodes, but we must take into account that:

- hundreds or thousands of non-striped files distributed among all the storage nodes allow Active Storage to exploit the processing capacity of all of the nodes.
- if we have a few files, these must be striped across the storage nodes if we want to use all the nodes for the Active Storage job.

There are also some considerations with respect to the implementation of the processing components, specially when files are striped. If files are not striped, the processing components can be implemented as any other sequential program. The use of big I/O requests can improve the performance, but the readahead mechanisms of Lustre and the Linux kernel can provide the same benefits with small requests.

When files are striped, however, the readahead must be avoided. Otherwise, a processing component reading a file on one storage node can cause the reading of data blocks from the storage node which has the next chunk of the file. This causes network traffic, which can hurt the performance. The readahead should not be disable on a system-wide basis because the overall system performance can be downgraded. It is better to disable it on a per process basis. However, in doing so, the performance of the processing components can also be downgraded.

One approach to avoid the readahead and get a good performance at the same time, is to make the processing components read every local chunk in only one I/O operation. That means that, if the stripe size is 1MB, every processing component must read its local data in 1 MB read requests. Since a storage node does not have consecutive chunks of a file, the process will produce a stride access pattern, which will turn the readahead off.

When dealing with striped files, take into account that the same file will be processed by several processing components on different storage nodes. That means that the processing components must not truncate an existing file while opening it, because if a processing component has already written some data to the file when another processing component is opening the file, the data will be lost. For transparently-accessed striped files, the Active Storage library detects this incoherency and refuses to open the file.

If you decide to use `stdio` to manage files, also note that “r+” is the only valid open mode to access the striped files that you want to write. “w” and “w+” truncate the file and could produce data lost, as we have described before. With “a” and “a+”, writes *always* occur at the end of the file (even if you modify the file pointer position with `fseek`). This is not what we want when dealing with striped files because every processing component must be able to write to a given offset inside the file. For transparently-accessed striped files, the Active Storage library also checks the open mode of the files and refuses to open a file if the open mode is not “r+” when the file is to be written.

10. Installing Active Storage

The Active Storage framework is a set of Python programs. Therefore, Python needs to be installed in our system. Since `asmaster` runs remote commands by using `rsh`, this service also needs to be enable in the storage nodes.

The following instructions install Active Storage on a cluster with a Lustre file system:

1. Setup a Lustre file system where the OST nodes must also be clients of the parallel file system, i.e., they must mount the filesystem locally. Let us assume that all the OST's mount Lustre on `/lustre`.

2. Untar the active-storage tarball file in a globally accessible directory. Any directory in an NFS file system would be fine, but, since we are using Lustre, we recommend to untar the archive in the parallel file system itself. There must exist a `/lustre/asd` directory after untarring the tarball.

3. Enter to the `/lustre/asd/libas` directory and run "`make`" to build the Active Storage library.

4. Add the `/lustre/asd` path to the PATH environment variable.

5. The node where you run the `asmaster` program must be one of the clients of the parallel file system. Our current implementation remotely launches the different processing components by means of the `rsh` command, so make sure that this node can remotely access to any storage node by using `rsh`.

Finally, create a rule, save the corresponding file in the `/lustre` directory (or any other directory underneath) and run the Active Storage job by entering, for example, "`asmaster /lustre/rule.xml`".

11. Rule syntax's reference

The complete syntax of a rule is as follows:

```
<?xml version="1.0"?>
<rule>
    <stdfiles>
        <stdin>file (/dev/null by default)</stdin>
        <stdout>file (/dev/null by default)</stdout>
        <stderr>file (/dev/null by default)</stderr>
    </stdfiles>
    <match>
        <pattern>file pattern</pattern>
        <from>source for the @ string (* by default)</from>
        <to>@ string(* by default) </to>
        <trigger>yes|no (no by default)</trigger>
        <numprocs># (-1=infinite by default)</numprocs>
    </match>
    <program>
        <owner>user.group (asmaster's user.group by default)</owner>
        <path arch="any">program path for any architecture</path>
        <path arch="i386">program path for x86 nodes</path>
        <path arch="ia64">program path for ia64 nodes</path>
        <path arch="x86_64">program path for x86_64 nodes</path>
        <arguments>list of arguments (empty by default)</arguments>
        <perchunk>yes|no (no by default)</perchunk>
        <delete>files to delete (empty by default)</delete>
        <getstripe>list of files (empty by default)</getstripe>
    </program>
    <mapper>
        <path arch="any">mapper path for any architecture</path>
        <path arch="i386">mapper path for x86 nodes</path>
        <path arch="ia64">mapper path for ia64 nodes</path>
        <path arch="x86_64">mapper path for x86_64 nodes</path>
        <arguments>list of arguments (empty by default)</arguments>
    </mapper>
    <time>
        <start>start time (1970-1-1 0:0:0 by default)</start>
        <end>end time (2038-1-1 0:0:0 by default)</end>
        <kill>yes|no (yes by default)</kill>
    </time>
    <filesystem>
        <type>lustre|pvfs (lustre by default)</type>
        <mntpoint>mount point (empty by default)</mntpoint>
        <extranodes>list of extra nodes (empty by def.)</extranodes>
        <numprocs># (-1=infinite by default)</numprocs>
        <striping>virtual striping (1:1048576 by default)</striping>
    </filesystem>
    <logfile>log file (stdout/stderr by default)</logfile>
</rule>
```

The meaning of every element is:

`<stdfiles>`:

- `<stdin>`/`<stdout>`/`<stderr>`: files to respectively redirect the standard input, standard output, and standard error output of the *processing components* (a processing component is every instance of the program to run in every node; see the path element below). By default, inputs and outputs are redirected to `/dev/null`.

`<match>`:

- `<pattern>`: a file regular expression that specify which files `asmaster` must care about. For example, `/lustre/molecule*.in` says that `asmaster` must run one processing component (or more, see below) per every file whose name matches that pattern.
- `<from>` and `<to>`: by means of these two elements, we build the string which the symbol @ represents. For example, suppose that `<pattern>` is `/lustre/molecule*.in`, `<from>` contains `*.in`, and `<to>` is `*-chem`. If the file `/lustre/molecule240.in` is created in OST0, the @ symbol in that OST will be equal to `/lustre/molecule240-chem` because `<from>` specifies that the suffix `.in` must be removed from the filename, and `<to>` says that the suffix `-chem` must be added to the string produced by `<from>`.
- `<trigger>`: indicates whether we must wait for the creation of new files which match the pattern (`yes` value), or create a processing component for every existing matching file and exit (`no` value).
- `<numprocs>`: the maximal number of processing components which can be created in the same node. The default value is -1 (no limit). Note: if a list of extra nodes is specified (see the `<extranodes>` element below), and one of the OST's appears in the list, then the number of processing components in that node can be as high as the sum of the `<numprocs>` values which appear in the `<match>` and `<filesystem>` elements.

`<program>`:

- `<owner>`: user and group of the processing components. By default, they are the same as those of the `asmaster` process.
- `<path>`: path of the program to run on each node (usually, an OST). If the nodes belong to different computer architectures, the "arch" attribute allow us to give a different path for every architecture.
- `<arguments>`: list of arguments to pass to the program.
- `<perchunk>`: this option causes the creation of a processing component for every chunk of the input file. The option also causes the definition of the environment variables ABSCHUNKOFFSET and ABSCHUNKNUM (see **Variable Expansion** above).
- `<delete>`: list of files to delete after the program execution. The list can have file regular expressions.
- `<getstripe>`: comma-separated list of files. The striping information for every file in this list will be stored in the environment variable STRIPING and passed to the processing components (see **Striped Files** above).

`<mapper>`:

- `<path>`: path of the mapper. It is possible to have several paths for different computer architectures.
- `<arguments>`: list of arguments to pass to the mapper.

`<time>`:

- `<start>`: the time when the actions specified by the rule must be performed. If no value is specified, the processing starts immediately.
- `<end>`: the end time. Beyond this time, no further files matching the pattern are taken into account. If no value is specified and `<trigger>` is `yes`, the rule is active indefinitely, and the `asmaster` program must be interrupted manually. When `Ctrl+C` is pressed, the remote processes are gracefully ended, and the existing files still not processed are shown.
- `<kill>`: this element indicates whether the processing components must be killed if they are still running when the end time arrives (`yes` value), or not (`no` value).

`<filesystem>`:

- `<type>`: parallel file system to use (`lustre`, `pvfs`, and `localfs`).
- `<mntpoint>`: directory where the parallel file system is mounted on. This element must be specify when the file system is PVFS2.
- `<extranodes>`: comma-separated list of spare nodes which can be used to run processing components on. These nodes are picked up only when a new matching file must be processed, and the corresponding storage node has reached the limit of processes to run. Files to be created, if any, will be created with the same layout (striping) policy as if the processing component is run in the storage node. I/O will not be local, but it all will be redirected to the

same storage node.

- `<numprocs>`: the maximal number of processing components which can be created in the same node extranode. The default value is -1 (no limit). See the description of the `<numprocs>` element of `<match>`.
- `<striping>`: virtual striping for files stored in a local file system. By default, the striping is 1:1048576 what means that there is only one storage node and that the stripe size is 1MB. Because there is only one storage node, this is equivalent to have non-striped files.

`<logfile>`:

- File to store the output of the `asmaster` program. By the default, the standard output and standard error output are used, as necessary.