Heterogeneous distribution in ATLAS

D. Sánchez-Crespo, M. Fairén and À. Vinacua

Department of Software, U.P.C.
Diagonal 647, 8^{ena} planta
E08028 Barcelona, Spain

April 6, 1999

Abstract

The ATLAS platform allows unsophisticated programmers to include advanced features in their applications with no or very little extra information and effort. These features include network distribution of cooperating processes, a powerful macro-language, a flexible journaling system and some other mechanisms directly related to graphics applications problems. In this report we describe how ATLAS is able to distribute the application processes among different architectures without needing a previous configuration information of the available processes for each architecture.

1 Introduction

ATLAS is a software package designed to allow users to share resources across local area networks. Its main features include transparent distributed execution of processes, IPC (inter-process communication) mechanisms, as well as journaling and other specific services (see [3] for a more generic description).

As ATLAS is a multiplatform environment, supporting features that ease the execution of processes over heterogeneous local area networks, one important feature is transparent execution of processes, which allows a user to run a process in the network by simply knowing its name, regardless of:

- which machine(s) have the desired process available
- which machine is the user connected to

ATLAS selects from the different machines that can execute the program the one that better accomplishes the task. In fact, although it is not implemented in the version this document presents, one of the ATLAS aims is to make a load

balancing of processes among the network. The user's view will be that of a virtual machine he talks to (the ATLAS system) which will encapsulate and hide all the complexity, but also offers all the power, of the network.

2 Global architecture managing the distribution

Before we enter in details concerning specific mechanisms, it may be useful to give an overview of the generalities of the ATLAS kernel. It has two components: servers and `distr`. As ATLAS is a distributed system, it requires a component to be used on a per-host basis, which defines that host as an ATLAS-capable machine. This component is the server (the `server_atlas` process). Servers do not communicate between them, but only with the `distr` process, which gives them orders to execute specific processes for the application.

Second, we need another piece of ATLAS to manage the distribution and give the execution orders to the servers. This piece is the `distr` process. The name `distr` (distributor) is because one of its main tasks is the distribution of the processes to the different hosts (thus, the servers) available.

There are also some configuration files which are important to several mechanisms ATLAS uses for the distribution. These are:

AtlasSettings This is a user's configuration file. It must be located in the user's home directory, under the name **$HOME/.AtlasSettings**. This file configures two things, the directories where a process can find shared libraries and the hosts allowed by the user to be used by ATLAS for his applications. We will see its structure in more detail later.

AtlasConfig This is an ATLAS internal configuration file. It should be located in the ATLAS installation directory, under the name **bin/AtlasConfig**. It must be changed only under special circumstances, and always by the ATLAS system administrator. It is used by both the `server_atlas` and also the `distr` processes.

ATLuserid Each ATLAS user has a unique identifier that prevents faked identities. This identifier is contained in the file **$HOME/.ATLuserid**, and is a 32-byte hash of random information from the system at the time of creation (essentially 32 random bytes) that ATLAS will use for all user identification. In this way security is guaranteed. The file is used by both the `server_atlas` and the `distr` processes. It must be generated the first time a user runs an ATLAS application by using the "`genid`" utility also included in the ATLAS installation files.

2.1 The `server_atlas` process

The `server_atlas` process is a daemon which should be running in each of the hosts willing to provide ATLAS execution services. It is connected to the `distr` process by a connectionless protocol (meaning each command is self-contained and there are no command sequences) and it is running with root privileges. The server implements as of today two main functions, a broadcast response and a process execution. We will see them later when discussing the corresponding mechanisms.

Keep in mind that the `server_atlas` process is stateless, which means that it does not keep internal data structures permanently. In other words, each command processed by the server is independent, and it implies starting from zero, performing some tasks, and returning to the starting point. This fact is important to understand the internals of the `server_atlas`.

2.2 The `distr` process

The `distr` process, on the other hand, is more complex. We will see in this report that it manages the broadcast mechanism, keeps the needed information of each process available for the application and decides the host where a process must be executed depending on availability of the process in the host. But the `distr` process is also the center of every interprocess communication in an ATLAS application and also controls other mechanisms important for those applications like journaling, fault-tolerance to crashes or disruptions, asynchronous user input data, etc. (You can see [2, 3] for a more general description of the whole ATLAS system.)

The `distr` process is the process the user executes in order to start an ATLAS working session. It has an execution cycle similar to the rest of the ATLAS processes (such as `server_atlas`). They work in a per-event basis: they do nothing but waiting for requests from the processes connected to them (normally `distr` is connected to all the others). When one such event arrives, the process "wakes up", processes the request and sleeps until another request arrives. `distr` is somehow analogous, but it has a more complex start up process because it must control most of the ATLAS mechanisms, including the broadcast and the process execution mechanisms.

2.3 The ATLAS "Sysid" to recognize architectures

To make ATLAS able to find a process across the network and execute it, obviously it is required some setup work and following certain guidelines.

First of all, ATLAS does not scan whole hard disks looking for processes to execute. Whenever a user wants to execute a file with ATLAS, he or she must

put it in a special directory. The key items to access this feature are System Identifiers, *Sysid*.

An ATLAS SysId (System Identifier) is an ASCII string used by ATLAS to group together several binary-compatible machines in a local network. This means that all computers sharing a unique SysId must be able to execute the same binary files. For example, two PCs can execute exactly the same files, because they share the same processor architecture. Thus, their SysId should be the same.

The SysId is therefore not a computer identifier, but a "binary architecture" code. It is built automatically by the ATLAS system, and has the following format:

```
sysname-release-machine
```

where each substring (sysname, release and machine) are derived from the `uname` system call. In fact, writing on a system shell the command `uname -srm`, it will print the three members, and in the desired order. Sysname identifies the family of computers your system belongs to. For example, `"SunOS"` would identify Suns, as `"Linux"` would identify Linux based machines. The second term, release, is used to identify what version of the OS is the system running. The uname call will return a full version number, such as `5.5.1`

As far as ATLAS is concerned, only the major release number is required, so any minor version number or additional information will be stripped away. Finally, the machine identifier is a vendor-dependent code which identifies the hardware implementation. For example, some Silicon Graphics systems have machine code IP25, and some others have the IP32 code.

There are, however, some observations to be made. First, some machines have sysnames that are mixed-case. `"SunOS"`, for example, would be such case. The ATLAS SysId, following the naming convention of Unix, is case-sensitive.

Second, there are some machines with OS release numbers containing non-numerical information. Some HP machines exhibit this behavior, having release numbers such as B.10.10. In these cases, ATLAS will consider that the major release number is the first number found scanning the release number from left to right. This means that in our example "B" will be dropped and the version number would thus be 10.

Last, but not least, sysnames and machines can contain some special characters, such as slashes, points, or other printable characters. ATLAS strips these characters away from the names, and so machine names such as `"9000/735"` will be converted to `"9000735"`.

To avoid problems deciding which is the correct ATLAS Sysid for a given architecture, ATLAS comes with a handy utility which will tell you the SysId of any machine. This program is called gensysid, and may be invoked at any time from the shell command line by writing:

```
gensysid <CR>
```

and its output will be a message telling you the desired string identifier.

When the Sysid is used by the developer to identify binary–compatible architecture directories, this ATLAS Sysids can be also wildcarded, thus allowing a unique identifier to be shared by different architectures. This adds flexibility to the system. For example, a Sun machine could have Sysid `"SunOS-5-sun4u"` which would mean a Sun UltraSparc with Solaris and a different machine could have the Sysid `"SunOS-5-sun4m"` which would mean an older Sun machine, but also with the Solaris Operating System. Now, there is no need to use two different identifiers, as both machines are binary compatible. Thus, it would be useful to allow Sysids to define which of the three fields (vendor, os, and machine) should be taken into consideration, and which should be simply ignored. The way to achieve this results is by using the wildcard `"any"` as a substitute of the undesired field. For example, the two Sun machines described earlier in this section had Sysids `"SunOS-5-sun4u"` and `"SunOS-5-sun4m"`. So, a well-constructed global Sysid for them both would be `"SunOS-5-any"` as we consider the machine field to be irrelevant.

In case the ATLAS daemon (`server_atlas`) is also working in a not binary compatible machine also working with Solaris (SunOS-5), for example a SunOS-5-X86, the distinction should be done for this machine in order to avoid a process compiled for the others to be sent to this machine.

3 The broadcast mechanism

Every time a `distr` process is started somewhere in the network, it issues a broadcast message to the network where some servers will be in turn listening to these kind of messages. This message is thus called Distr Broadcast Message, and has the following structure:

bytes	**containing**
0..3	length of the message (network formatted int)
4..7	user identifier of the user executing ATLAS (net formatted int)
8..11	group identifier of the user executing ATLAS (net formatted int)
12..k	*Sysid* for `distr`. Variable length string
k+1	Null-terminator character
k+2..l	Execution path for `distr`. Variable length string
l+1	Null-terminator character
l+2..m	ATLuserid of the user executing ATLAS
m+1	Null-terminator character

This broadcast message tells the servers who (uid and gid) is running the newly-created `distr` process, which machine architecture (Sysid) is the `distr`

using, where (absolute pathname) has been the `distr` started, and finally the ATLuserid of the user who started ATLAS.

3.1 `distr` sending the message

When the `distr` process is started up it must send the broadcast message in order to find out the information of the hosts available to execute ATLAS processes. Before sending the message it must do some initializations. These are:

- **Port customization:** ATLAS does not require specific ports to work. In fact, the ATLAS system manager may specify which ports (two are required) will be used throughout the execution by changing the contents of the AtlasConfig file. This file may only be changed by the ATLAS system manager, and specifies vital information for ATLAS. This file should only be touched when strictly needed, and can be found in the bin directory of your ATLAS distribution. The only lines needed to customize the ports are:

  ```
  $PORTS
  stream = 5021
  datagram = 5023
  ```

  Here you can change the values of any two ports (not necessarily contiguous). The only requirement is that both ports must be free. You may check this with the netstat command.

  This port customization, of course, is also done by each `server_atlas` process when they are started. This is usually done by the ATLAS system manager at the installation time.

- **Process–Host Table initialization:** Before receiving any server response to the broadcast message, the Process–Host Table is created. First the AtlasSettings file is scanned in order to find the allowed host list. This list is a part of the .AtlasSettings file (located in the user's home directory) and is not mandatory. It is useful for those users that want to "ban" hosts from the network. The syntax is:

  ```
  $HOSTS
  host-name1:host-name2:...:host-nameN
  ```

  If the ATLAS `distr` process finds a record like this it will only treat those responses sent by servers running over the hosts in the list. As we said previously, this node of the .AtlasSettings file is optional. So, if not present `distr` will process all messages from servers that respond to the broadcast message, without further filtering.

After these initializations, the `distr` process is ready to send the broadcast message to the network. Thus, the message is sent, and also a handler is attached to the "ACE Reactor" object [5] to easily detect the responses coming from the servers. A timer is also added into the Reactor so that we give some time for the servers to respond. In this first prototype `distr` only listens during this given time interval. This interval has the default value of 3 seconds, and is built into ATLAS with the variable `DEFAULT_BROADCAST_TIMEOUT` defined in the 'globals.in' file. However, the ATLAS system manager can customize this value to specific needs. For example, on a very fast and reliable network maybe 1 second of timeout will do. On the other hand, a slow network may need more time. It can be customized by changing a parameter into the AtlasConfig file. The node to be changed is:

```
$BROADCAST
timeout = 5
```

and the substitution must be on the number of seconds of timeout. Remember that if the default 3-second delay is enough, there is no need to have this node on the AtlasConfig file.

To sum up, the `distr` process generates the broadcast message, sends it, and waits for responses to arrive within a specified time interval.

3.2 Response construction in `server_atlas`

We can see the broadcast processing divided in four steps:

1. **Message reception.** Let's now see how the broadcast message is received and processed by a `server_atlas`. Servers keep two sockets open throughout their execution cycle. One of them is a datagram socket, and is used only to process broadcasts and send responses to them. The read function (`BCastRcv::ReceiveBCast`) peeks at the socket when it is notified that there is data available to read, and then consumes it. Then, the different fields of the message are extracted (function `BCastRcv::DecodeBCast`).

2. **User authentication.** First, let's remember that servers must be available all the time to process requests (either other broadcasts or execution requests). So, it is mandatory that steps two (user authentication) three (response generation) and four (response emission) do not block the server. Thus, a sub-process is created, and all processing is performed concurrently by the main and secondary threads: one taking care of the sockets and waiting for requests, and the other performing steps 2, 3 and 4 of the broadcast processing. After message reception, comes the second part, which is the user authentication. ATLAS attempts to prevent situations such as:

– faked identity by the poster of the message
– non–registered user

by performing simple verifications. These verifications consist of:

- First, the broadcast message contains the uid and gid of the user. It also contains an ATLuserid code, which is filled by the `distr` process with the code for that user. An illegal request could probably fake the uid and gid of a certain user, but faking also the ATLuserid would be more difficult, as this information is contained in a file readable only by its owner.
- Second, if an illegal request arrives with a uid, gid and ATLuserid, the server will access the information on the ATLuserid for that uid and gid, and check if that ATLuserid is the same we were sent by the socket. The way to do this is to access the /etc/passwords file with the command `getpwuid`. This file contains (among other information) the home directory for every user. So, given the uid and gid we can access his home directory, perform a setgid and setuid and read his .ATLuserid file. This way we perform the verification of all data sent being correct.

This provides user authentication as secure as the user's file system and network. The network security could be improved using cryptographic protocols, which is not done at present. It seems however unnecessary to provide stronger security than the user's file system under any circumstances.

Once this step is completed, the server knows the request is valid, and is thus ready for the central step, which is the response generation.

3. **Response generation.** The response of a broadcast message is a list containing the following information:
   – Host name (server which generated the response)
   – Server's Sysid
   – Available process list

   The first two fields are straightforward: the `distr` process needs to know who is answering the broadcast message and what architecture does the responder have. The third element is a list containing all the files that the responding server found on its directory scan. We will now explain the exact contents of this list, and its creation algorithm.

   Whenever a server replies to a broadcast message, it scans certain areas of the file system searching for executable files. Once done, it returns that file list to the `distr` process. This way when the user asks `distr` to execute a file, it is easy to check which hosts have that process available for execution.

   This list is priority–ordered. This is useful whenever a process in available in two different places of the disk. For example, with the following structure:

```
$HOME
      Atlas
             bin
                  SunOS-5-any
                       process1
                  IRIX-6-any
                       process2
                  any
                       process1
                   (...)
      (...)
```

we have the process "process1" available at two different locations: the specific SunOS-5-any and a generic one located at "any". Logically, the first instance should be prioritized, as it refers to a more specific location. Thus, in the executable file list which will be returned to the distr process the process $HOME/Atlas/bin/SunOS-5-any/process1 should be before the same name process located at "any". The way to build such a list is scanning the selected areas of the disk in importance order: most important ones first, secondary after. So, the algorithm for scanning the disk is:

- Scan the area of the disk around the path where distr has been started. If that path is inside an architecture directory (such as the path $HOME/process/bin/SunOS-5-any, we will first locate the directory corresponding with the server's Sysid, and then scan it. After we have scanned that directory, we will scan the several wildcarded versions (first wildcarding machine, then OS, and finally sysname). Finally, we will scan the "any" directory and also the base directory. So, the order in which directories are checked is: (assuming the Sysid is in the form S-R-M)

```
i)   ..../S-R-M
ii)  ..../S-R-any
iii) ..../S-any-M
iv)  ..../S-any-any
v)   ..../any
vi)  ..../
```

- Scan the $HOME/Atlas/bin area, including the architecture directories for the server. Again, the order of checking is:

```
i)   $HOME/Atlas/bin/S-R-M
ii)  $HOME/Atlas/bin/S-R-any
iii) $HOME/Atlas/bin/S-any-M
iv)  $HOME/Atlas/bin/S-any-any
```

```
v)   $HOME/Atlas/bin/any
vi)  $HOME/Atlas/bin/
```

Note how we scan this `Atlas/bin` section after we scanned the region around the `distr` boot path. This behavior is intentional. As we said in preceding sections, placing executable files in the same path where we start the `distr` process is a convenient way of working with ATLAS while developing and testing new code. Thus, in case we have the same process in the starting path and in one of the `$HOME/Atlas/bin` sections, we wish to use the first one as the preferred.

- Third and last, we will scan the installation directories of ATLAS. These directories should contain only binaries related to the ATLAS distribution, such as `server_atlas`, `distr`, and several utilities. So, it makes sense to make this the last (less priority) location for executable files. Here again we will scan directories using the Sysid wildcarding feature as described in the two preceding steps.

For every file and directory we encounter during this phase, some tests are made to ensure proper execution. Then, for every FILE, it is considered executable if, and only if:

a) It is a file and has execution permission by its owner, or

b) It is a symlink, and the file linked by it satisfies **a)** step.

And for every DIRECTORY, it will be expanded if, and only if:

a) The process of scanning determines that directory as a possible location of files (a matching architecture directory, "any", etc.), and

b) The user has read and execution rights over that directory. No symlinks are accepted here in the current prototype.

4. **Response emission.** Once the scanning process is complete, we have a list of files, along with the Sysid and domain name address of the server which performed the scanning. Thus, it is now time to format this list into a network packet, and send it through the network to the `distr` process. So, we build a message with the structure:

   address + own Sysid + 1{process name}N

   (where i{x}j means repeat {x} from i to j)

   Between fields we use null characters (ASCII code 0) as delimiters, and end–of message is notified with two null characters in sequence. Albeit redundant, the message is preceded by a network–formatted integer value which contains the total length (this is, the four bytes for the integer and all the message data) of the message which is used by the driver to simplify the processing of datagrams.

Whenever the `distr` process receives a server response, it wakes up (specifically, the routine `BCast::handle_input` is woken up by the Reactor). This routine is fairly simple. First, it decodes the broadcast response which is received through the datagram socket and then the message is passed to the PHT (Process–Host Table), where it will be divided into its main elements, and inserted into the structure (if and only if the host that responded is allowed by the AtlasSettings file).

3.3 Process–Host Table

During normal `distr` operation, one of the main tasks to be carried out is to decide, each time the user wants to execute a process, which server is best suited (in terms of process availability and also performance) to execute it. The `distr` process has a data structure specially designed to aid in this complex decision, the Process–Host Table (PHT). The PHT is a variation of a multilist, as can be seen in the following scheme:

Host 1	X	X	X
Host 2		X	X
	Process 1	Process 2	Process 3

The above drawing tells us that Host 1 has Processes 1, 2 and 3 available, and that Host 2 only has processes 2 and 3. The above drawing would depict the classical multilist as found in countless data structures and algorithms textbooks. Our PHT works in a similar way, but adding priorities. In sections 2.3 and 3.2 we said users are allowed to store their binary files in different locations, and that ATLAS will scan the disk in order so more priority areas are searched before less relevant ones. So, we will use the following data structure, which is just a variation of the one explained previously:

Host 1	1{path}N	1{path}M	1{path}N
Host 2		1{path}O	1{path}P
	Process 1	Process 2	Process 3

Here, each Host–Process node keeps a list of all instances of the process in the host's file system. The list is priority–sorted, so the first element has the most priority, and so on.

This data structure can be found in the file "HostProc" of the ATLAS distribution, and it has two main relevant operations: inserting a new list of processes available in a particular host, and finding the most suitable host for executing a process.

The insertion routine

This insertion has the disadvantage (due in part to the nature of the broadcast system) that it does not insert host–process pairs, but lists in the form:

hostname sysid process0 process1 process2 ...

Process(i) are the processes available at that host. So, our routine reads the first two elements in the list (host name and Sysid), builds the Host Information block (named "InfoHost"), and thus opens a new "row" in our multilist, which is really a hash table to speed-up access. Now, for every process in the list, it appends it to the corresponding list. As the initial list is priority–sorted, we can ensure that the lists created will also be sorted.

The selection routine

As we have seen, the insertion routine is row-wise, or horizontal. The selection routine will be vertical, or column-wise. What we will do is scan from top to bottom each host in the host list, and keep track of the number of processes currently running at each one. Then, for each host, we seek the process we want to execute, selecting it from the first position of the corresponding list. Now, as we scan top to bottom, we will select a new host as the executor if and only if it has less processes currently under execution, but has the desired process available.

It is important to note some behaviors which may seem strange at first sight, but are fully normal:

First, two hosts (say H1 and H2) have the desired process, and H1 has less processes currently running (thus making it a best candidate than H2). Then, H1 will be chosen as the executor, regardless of where the process resides. For example, if H1 is best-suited, but has a process in a low-priority location, but H2 is a bit more occupied, but has a high-priority location process, H1 will be chosen. So, host choice is considered more important than location choice.

Second, ATLAS does not really support full load balancing by now. As of today, what ATLAS really does is keep track of how many ATLAS processes are being run and where. Thus, the decision criteria of "choosing the host with less processes" should be understood better as "choosing the host with less ATLAS processes". Keep in mind that this criteria is still too weak. For example, if H1 has 400 processes in run state (and 3 of them are ATLAS processes) and H2 has only 4 processes (all of them being ATLAS processes) and we perform a host selection for a process available at both locations, H1 will be chosen, as it has 3 ATLAS processes and H2 has 4. The different computing power of the hosts is not taken into account either at present.

4 A process execution

4.1 Deciding which host

Once the broadcast interval has ended, the PHT now has information about the processes and hosts available. It is time then to start the execution of ATLAS. The `distr` process has to start then the Command Subsystem process and send to it the ATL file having the initialization of the application (see [1] for more details of what an ATL file is).

Whenever a certain process must be executed the `distr` process uses the information contained in the PHT to locate the best server to execute this process, and will then try to communicate with it. The core routine here is `QuinHost` which, given a process name, scans the PHT to find the best possible host. The mechanism used by `QuinHost` has been explained in section 3.3, "The selection routine".

4.2 The actual execution

To execute a process, `distr`, having decided which host is the best one to do it, generates a message to be sent to the server on that host, ordering the execution of the process.

This message includes information about the environment that must be set to execute the process. But this information is the same for every process during the application execution, so `distr` stores it during its initialization.

The environment information is a list which includes the following information:

- User identifier (result of a `getuid` call)
- Group identifier (result of a `getgid` call
- ATLuserid value (from the .ATLuserid file)
- DISPLAY environment variable
- HOME environment variable
- PATH environment variable
- PWD environment variable
- ATLAS_ROOT environment variable (only if it is defined)
- LD_LIBRARY_PATH environment variable (only if defined)

All this information is kept throughout the execution cycle.

The message sent to the server then has the fields corresponding to the list above having the ATLAS_ROOT and the LD_LIBRARY_PATH as optional.

We will now focus on the events following the reception of this execution message by a given server, explaining the algorithm and tests it performs. This function can be subdivided in the following steps:

- **Message reception.** First of all, the server's "MessRcv" object is notified (via the `handle_input` method of the ACE library [4]) of the availability of data in the stream socket. Keep in mind that datagram sockets are used for broadcasting purposes only, and stream sockets are used for execution messages. The Atlas process execution message is a variable-length ASCII string, with some differences with the broadcast message which are worth noting:

  First, fields can be optional. The broadcast message is fixed-structure, variable-length. This new message is variable-structure (and of course variable-length).

  Second, there is no order in the sequencing of the fields. As far as the message contains the required information, this data can be ordered in many different combinations. This didn't happen with broadcast message.

  Once the message is read into a list of buffers, we reconstruct the several fields and create a list of parameters.

- **User authentication.** Before processing an execution request we must be sure that the sender of the message is really allowed to execute the process, and that no security violations can arise. So, we re-check the ATLuserid using the /etc/passwd file exactly the same way we explained in the broadcast section (3.2).

- **Environment setting.** Before executing the process, we must set its environment. The reason is simple. Remember we said the server process starts when the machine boots, and is always under execution as a daemon. So, it has the environment set at boot time. To ensure security, the process spawns a child process whenever operations are needed, thus keeping always an eye on the sockets. This child then performs a setgid and a setuid command, releasing its root privileges. Still, it lacks environment, which should be set in a per-command basis. Every time we want to execute a process, we will set some environment variables (PATH, LD_LIBRARY_PATH, etc.) accordingly, and then execute the process with the `exec` command. This way we ensure that processes are executed within their right environment and as user processes.

  The environment variables that are set are (in the following order):

  First, we set the variables received within the execution message.

  Second, we set the ATL_LOCAL_EXECUTION_PATH with the path to the executable file. As ATLAS executes processes with the `exec` call and it wants that the PWD environment variable sent by `distr` be inherited, the child process will not have access to the path where its binary file resides. So ATLAS must allow and alternate mechanism to allow child processes access to their directory structure. This is particularly useful whenever our binary file must access some data files in its installation tree. If we do not know the path where the process was executed, how can we access files located in paths relative to this one? (see also section 5.3).

The ATL_LOCAL_EXECUTION_PATH is used so user processes can access local files. Whenever a binary file starts (and wishes to access its disk data), it must get the value of the ATL_LOCAL_EXECUTION_PATH variable, and use it as its home directory.

Third, we set LD_LIBRARY_PATH adding to its value the libraries specified in the ".AtlasSettings" file. This file (located in the home directory of every ATLAS user) contains useful information. Among other data (see specific section), it contains which library paths should be considered when executing under any of the available platforms. For example, a file containing:

```
$LIBRARIES
SunOS-5-sun4u = /homes/husers10/atlas/danis/Atlas/Atlas/\
   lib/Sun5: \
   /usr/usuaris/sig/mfairen/proves_Atlas/ACE-4.5/\
   ACE_wrappers/build-Sun5/ace/:\
   /usr/local/tcl-tk8.0/Solaris/lib:  \
   /homes/hsoftsol2/Atlas/ACE-4.4/ACE_wrappers/ace/ \
   /usr/usuaris/sig/mdtl/nq/libbonsai/:\
   /usr/local/mesa2.4/SOLARIS/lib:\
   /usr/usuaris/sig/newdmi/nq/oscarsan/newdmi/lib

IRIX-6-IP25=/usr/usuaris/sig/mfairen/vonsai/Atlas/\
   build-IRIX6/lib:/homes/hsoftsol2/Atlas/ACE-4.4  \
   /ACE_wrappers/build-IRIX6/ace/

SunOS-5-sun4m=/usr/usuaris/sig/danis/atlas/Atlas/Atlas/\
   lib/Sun5:  \
   /usr/local/tcl-tk8.0/Solaris/lib:/homes/hsoftsol2/  \
   Atlas/ACE-4.4/ACE_wrappers/ace/

HPUX-10-9000735 = /usr/usuaris/sig/danis/atlas/Atlas/\
   Atlas/lib/HP10:  \
   /usr/local/tcl-8.0/lib:/usr/local/tk-8.0/lib
```

tells ATLAS that hosts with the Sysid `"SunOS-5-sun4m"` should check the following directories for libraries:

```
   /usr/usuaris/sig/danis/atlas/Atlas/Atlas/lib/Sun5
   /usr/local/tcl-tk8.0/Solaris/lib
   /homes/hsoftsol2/Atlas/ACE-4.4/ACE_wrappers/ace
```

So, these three entries will be prepended (added at the beginning) of the LD_LIBRARY_PATH we received through the network. As processes executed by calling `exec` inherit the environment of the caller, we can be

sure that the process being executed will be able to access its libraries correctly.

- **Sub-process execution.** Once all three steps have been completed, and no error has occurred, it is finally time to execute the binary file. ATLAS will only perform one final step: it will redirect the process' standard error and output channels to two files. This is only for logging and maintenance purposes. These two files are called (if the process is called "A"): "A.err" and "A.out", and will be in the same path the `distr` process has been started. As interaction with the ATLAS environment should be through the built-in graphical interface, these files are mainly used as log or debug info files of the process "A".

  After these have been opened, ATLAS will only execute the process with a call to `exec`, passing the following command line arguments into argv:

  `argv[0]:` full name (absolute path included) of the process

  `argv[1]:` process name (executable file name)

  `argv[2]:` OOB_Hack info. Used to prevent the Out_Of_Band Data error in Solaris 2.5 [1].

  And the execution will then overwrite the server's child process logical space with that of the desired process, and begin its execution.

5 What the ATLAS user must be aware of

5.1 Directories structure

As said in section 2.3, Sysids are used to refer to compatible machine groups. So, two binaries are compatible if they were generated in machines which share a common Sysid. The idea then is to organize binaries in a directory structure, using Sysids as their names. Sun machines will keep their binaries in a directory named with their Sysid, for example. This will prevent architectures from trying to execute code other than their own.

So, it seems now clear that users need to follow some directories structure guidelines in order to use the ATLAS multiplatform feature. To begin with, they should provide a directory called "Atlas" in their account's HOME. This directory will be used to store everything related to the ATLAS system. Then, this directory must have a subdirectory called bin, which will be the storage for ATLAS binaries.

[1] In Solaris 2.5 the `select` call doesn't wake up with only a 1 byte OOB data, so we adapt ATLAS communications to send in this case 2 bytes with the OOB data, this problem has been corrected in 2.7 (we have not checked 2.6).

This `Atlas/bin` directory can be organized in turn in a multiplatform structure, to allow the storage of binaries for several architectures. For example, a possible directory layout would be:

```
$HOME
       Atlas
              bin
                    SunOS-5-any
                    IRIX-6-any
                    any
         (...)
```

This structure would provide a place to store Solaris (Sun 5) files, and a different place to keep IRIX 6 files. These two folders do not need to check the machine member of the Sysid, as all possible variations can be considered compatible, so this part can be substituted by the wildcard "any".

Finally, the user has a directory called `any`, which will be useful to store generic binary files; shell scripts, for example.

The above explanation may suffice for most applications. Some situations, however, may require a higher level of customization. For such uses ATLAS provides an extended mechanism, which allows you to keep your executables in different places of your disk. For example, you may have a "stable" version of an executable file under a certain path, and a different copy (for example, a debug version) somewhere else. The guidelines to follow in this case are:

- Place the stable binary files under `$HOME/Atlas/bin`, as explained above
- Then, place yourself in a directory of your choice, and place binaries there. You may do so directly, or using a directory structure such as the one found around the `Atlas/bin` area. Now, start the `distr` process from this same directory. If you do so, `distr` will scan as the most-priority area the path you placed these new binaries in. For example:

```
$HOME
       Atlas
              bin
                    SunOS-5-any
                         proc1
       debug
              SunOS-5-any
                    proc1
              IRIX-6-any
      (...)
```

With the file structure described above, you can execute `distr` from the path `$HOME/debug` (this pathname is arbitrary). If you do so, the first executables it will scan for are the ones placed in that path in the corresponding architecture directory. So, if you ask ATLAS to execute the file `proc1`, it will always take as a first guess the file `$HOME/debug/SunOS-5-any/proc1`, assuming it has more priority than the version stored at the directories tree `$HOME/Atlas/bin/...` This criterion is intentional: `Atlas/bin` is a place to store stable files, whereas placing them in the `distr` initial path is a useful mechanism for debugging. However, if you still wish to give higher priority to the file stored at `Atlas/bin`, you just have to execute `distr` from a different path than the one containing the debug version.

5.2 The ".AtlasSettings" file

As we have seen in section 4.2, using only one definition of the environment variable LD_LIBRARY_PATH is not enough because we usualy will need different definitions for different architectures.

The ".AtlasSettings" file, as has already been described, should have the directories to be added to the default value of the variable for each architecture and it can also include a set of host names that will act as a filter to avoid using in the application other hosts (also able to run ATLAS processes). This file must be placed in the `$HOME` directory and configured by the ATLAS application developer.

5.3 How a process knows its relative path

Also in section 4.2 we have talked about an environment variable giving to the process being executed information about the path where its binary code is.

In order that the developer can use easily this information, ATLAS offers an API call which has the prototype

```
String atl_get_local_path ();
```

that gives a String which contains the absolute path for the directory containing the binary code for the executable process. Since ATLAS is able to decide depending on the architecture which directories look to search executable files, these executable files can be placed on directories different from the *current working directory*, then the process can access to relative paths (even though it doesn't know where its binary has been started) by prefixing this relative path with the result of the `atl_get_local_path` routine.

5.4 Debugging ATLAS proceses

One of the mechanisms used by ATLAS to offer *fault-tolerance* requires processes to regularly send messages to `distr` in order to notify their state. A process stopped in a debugger will certainly stop sending this "keep me alive" messages for a while, thus making the ATLAS `distr` process believe it is malfunctioning. In normally executed processes this would force `distr` to kill the user process, but it should not be applied to a process being debugged.

ATLAS introduces a mechanism which allows process debugging in a convenient way. A process being run can be entered in "debug state" at any time with a user command. From that moment on, the ATLAS `distr` process will keep in mind this situation, thus allowing the process to stop sending keep-alive messages. This way, the user can then attach a debugger to the involved program, and behave as if ATLAS really wasn't there, debugging normally. Once done, the process can be put again in "normal running mode". This restores its state, thus re-establishing the keep-alive message protocol.

The two commands involved in establishing and ending debugging sessions within ATLAS are:

```
atl_init_debug (string process_name)
atl_end_debug (string process_name)
```

They require the process name to be an already running process. Thus, to debug a process called "TestProc" with ATLAS, first, you would start the process with the command:

```
USE TestProc;
```

Second, you would start the debugging session by entering:

```
atl_init_debug ("TestProc");
```

There is, however, something to keep in mind. ATLAS is a multiplatform environment. A user process can be executed anywhere in your network, and you won't necessarily know where. The USE command will make ATLAS decide where to execute your process depending on the workload of each machine, and the availability of the given process across the network. So, when you start debugging your process, you will need the information about the host that is actually running your program, in order to attach a debugger to it. This information will be printed out for your convenience by the `atl_init_debug` command. This way you can find where you should place your debugger to work.

Once ATLAS knows you want to debug a certain process (by means of the init_debug call), it is time to effectively debug it. Here we will use gdb as the debugger, but you may use the debugger of your choice to accomplish this task. Remember you need to execute gdb at the host currently executing your process. In our example, we would do:

```
gdb
attach [process_identifier]
```

You need to know the pid of the process to debug. To find it, use for example the ps shell command at the machine executing the process (you know which machine it is because `atl_init_debug` gives you this information). Once you enter the attach command, your process is fully debuggable. You may set breakpoints, examine data, and perform step-by-step execution.

Once you have finished, you should restore the process state back to normal. This means you have to first detach the debugger from the process. Do this by entering:

```
detach
```

After doing this, you just need to issue the `atl_end_debug` command to end your ATLAS debugging session, and revert the process' state to normal. Do this by simply typing:

```
atl_end_debug([process_name]);
```

as an ATLAS command, and this will end the debugging session.

6 Conclusions and future work

We have described how ATLAS is able to distribute the application processes among different architectures without needing a previous configuration information of the available processes for each architecture. This information is automatically found out by ATLAS at its starting time. Although in the currently implemented prototype this information is only checked once at `distr` starting time, it is not the definite treatment for this mechanism which in future versions will be able to dynamically check this availability.

Another extension not yet implemented (as has been said before) is the real load-balancing mechanism. This mechanism will be the base for the decision of the host where a process should be executed.

References

[1] M. Fairén and A. Vinacua. ATLAS. Sistema de Comandes: Manual tècnic (in Catalan). *Report LSI-95-11-T*, 1995. http://www.lsi.upc.es/~ mfairen.

[2] M. Fairén and A. Vinacua. ATLAS, a platform for distributed graphics applications. In *Proceedings of Eurographics Workshop on Programming Paradigms in Graphics*, 1997.

[3] M. Fairén and A. Vinacua. ATLAS: a platform for transparently developing distributed applications. In *Proceedings of the Tenth IASTED International Conference on Parallel and Distributed Computing and Systems*, pages 467–471, 1998.

[4] D. C. Schmidt. The ADAPTIVE communication environment: Object-oriented network programming components for developing client/server applications. In *12th Sun Users Group Conference*, 1994.

[5] D. C. Schmidt. Reactor: An object behavioral pattern for concurrent event demultiplexing and event handler dispatching. In *Proceedings of the 1st Pattern Languages of Programs Conference*, August 1994.