The Report on HYDRA

G.H. Rolf

Computer Science in Chemistry and Physics

W.J. Melssen

Computer Science in Chemistry /

Laboratory for Analytical Chemistry

ABSTRACT

The HYDRA package features the control and synchronisation of parallel applications in a network with workstations. By moving applications from slower to faster hosts and automatically restarting an application when a host went down, HYDRA provides the flexability and fault-tolerancy required for long running calculations.

This document describes the break-down of conventional applications into parallel applications to be managed by Hydra, the features and tunable parameters of Hydra and all the user should know before getting started.

RFC: request for comment / draft version of 7 mei 1993

1 Introduction

One way of gaining computerpower is to bundle the power of a number of relative small computers. It is found that a certain class of computational applications can be split into identical parts that can be executed independently on a number of workstations interconnected by a local area network sharing a common filesystem.

This document describes the type and structure of scientific applications that can be run on multiple workstations in parallel. The means of control is provided by a software package called "*HYDRA*", which can start, move and synchronise the parallel applications.

HYDRA is called after a huge fire breathing dragon in the Greek mythology. Hydra is claimed to have nine heads, one of which is immortal. When one of the other heads is chopped off, it's replaced by a new one immediately...

Using HYDRA gives the programmer of an application the means to break the total work down into small pieces. After designing the splitup in parallel pieces, the programmer restricts the programming effort to just one piece. HYDRA provides the parallellism, synchronisation and continuation to run this small piece in parallel and in repetition to cover the required iterations of the total work to be done.

HYDRA's features include: moving applications to faster machines when performance has decreased, acting on hosts going down for any reason, suspending execution when no hosts are available with a low load or automatic suspension during office hours.

This document describes an application model that can be used for applications to run in parallel, the functions of HYDRA, its parameters and options and its display and logging facilities.

2 Application model

Many scientific applications read parameter files in order to find out what the user wants. Parameter files contain the values of certain constants, the range of values to use for the iteration, names of datafiles and so on. After reading the parameter file the application knows what to do and starts the calculations.

Long running applications need to be restartable. You may expect it to be aborted by an operator or by any incident on the machine it's running on.

Applications can be made restartable by defining steps in the total work to be done and to update files which contain intermediate results after each step (checkpointing).

Figure 1: Long running applications can be split into steps

The applications which are to be controlled by HYDRA use a slight extension of the stepwise checkpointing as described above.

- At the end of each step the application updates the datafiles containing the intermediate results and it updates the parameter files to reflect the parameters needed for the next step in the calculation.
- At the beginning of each step the application reads the parameter file. In other words: the contents of the parameter file determines the exact work to do in the step.

This scheme, which has some impact on the overhead, allows HYDRA to move applications to other hosts in the network. In fact the application that has to be moved is stopped after completing the updates and a new *instance* is started at another machine.

HYDRA can control a single application as described above, but there are applications that can be run in parallel. This means that iterations or *steps* can be calculated independent of others.

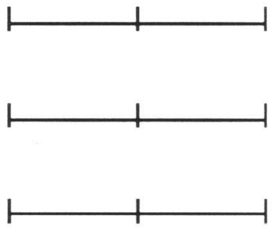

Figure 2: Independent steps can be run in parallel

There are several cases where independency can be achieved:

- subset of input
  Each parallel step can read part of the total input data that has to be processed.

- merge results afterwards
  The intermediate results of each parallel step can be combined into an intermediate result by merging the results of each step.

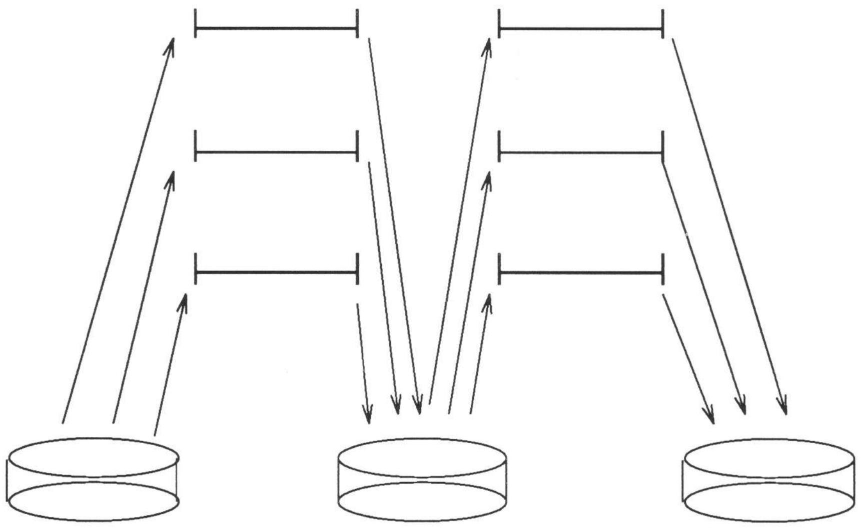

Figure 3: Each step reads data before and writes data afterwards

It's fairly obvious that lots of computational applications cannot be redefined into this parallel application model. In fact the model is related to the characteristics of *loosely coupled processors in a network*. Close interaction, e.g. in each loop of an iteration, is to be avoided as its costs in performance and network traffic are too high. In other words: parallel applications should run in parallel without interaction for a considerable amount of time.

2.1 The basic cycle

In this chapter we'll discuss the basic cycle that is defined for applications. Each time an application performs a step on a machine the following cycle is performed:

- read
  The application reads its parameters from a diskfile. In some cases it also reads the data it needs to perform the calculations.
- calculate
  The application performs calculations on input and produces output as results. In some cases the input is read from disk and the output written to disk immediately. Other applications keep the results in memory.
- write
  The output of the completed calculations is written to disk. This may include a merge of the total results of all parallel steps done. The application updates the parameter file to reflect the parameters for the next run.

We will call the distinct phases of the basic cycle *read stage*, *calculate stage* and *write stage*.

Applications communicate with a central process that controls and directs the activity of all parallel applications. In general, application processes receive messages which direct them to perform one of the *stages* of the basic cycle, described above. When a stage has been completed, the applications send a message to the controlling process to confirm.

During the *write stage*, all applications get the **write** command one by one, which provides the synchronisation of the update of the global file. This implies that after completing the updates of the global file applications should explicitly *close* the file. As a result local buffers will be flushed and the file will be consistent and complete to other machines in the network.

Optionally the *write state* may be 'simultaneous'. In that case you don't need the synchronisation and all applications get the *write* command at once. This allows each of the parallel applications to update individual datafiles in simultaneously.

3 Overview of HYDRA's functions

Before discussing the features and functions of HYDRA, we'll look closer at the architecture of the HYDRA package. The controlling program, called **mcp** is assisted by a process called **rups**, which collects the status of all hosts in the hostfile periodically. Both **mcp** and **rups** run on the same host, but all the parallel applications run on different machines:

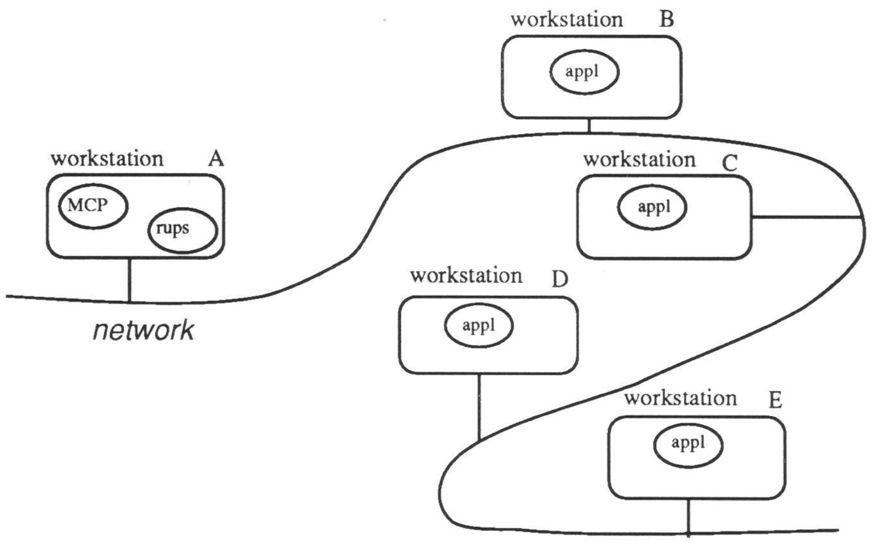

Figure 4: Mcp/rups and applications run on different machines in a network

Communication between *mcp* and the applications (*'appl'*) is implemented using TCP/IP interprocess communication *sockets*, which provide bidirectional reliable bytestreams.

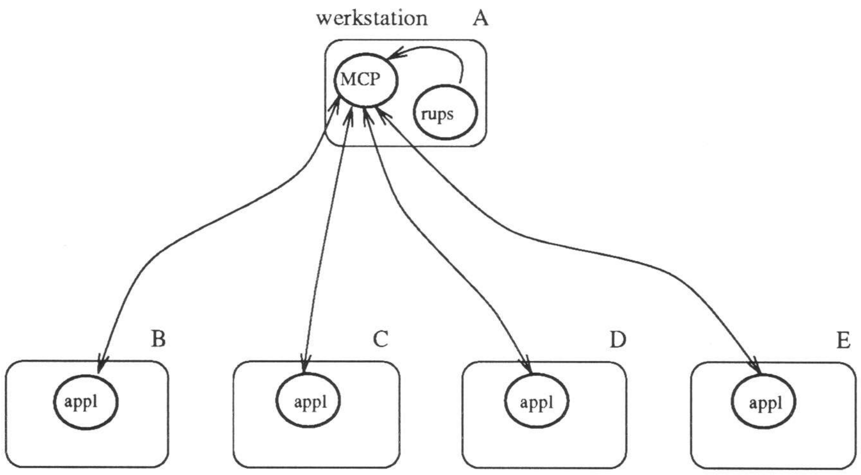

Figure 5: TCP/IP sockets provide the communications between mcp and the applications

We'll discuss the way applications are invoked remotely later in this paper. For this moment it is important that the application itself does not need to do anything to initiate the TCP/IP socket to *mcp*.

The functions and features of HYDRA are:

- The number of parallel instances is user defined and starts from 1. So even a single application can be directed by HYDRA.
- Each instance of the application program is started in a unique directory providing local files, private to the instance. As all hosts share the same filesystem, the parent directory may contain global files shared by all parallel instances.
- Applications may be run using a user defined *nicelevel.*
- There is a list of hosts available to **mcp** for this application.
- The **rups** process polls the hosts in the hostlist and obtains the *loadfactor* of each host. This number is the mean number of active, running processes during the past period. When a round-trip of all hosts is done, **rups** sends a single message to **mcp** containing all the *loadfactors.* **Rups** repeats this cycle after waiting some time.
- On receiving the table with loadfactors, **mcp** rebuilds the list of not-used hosts by order of *load and relative performance.* This will be discussed in more detail later. The host that is expected to perform the fastest is in the head of the list of free hosts. Hosts that did not respond to the polling of **rups** are reported "*possibly down*" only when their *loadfactor* is missing for the second time. In case such a host is being used, the application is restarted at another host. In most cases this is appropriate to recover from hosts that went down. During the *write stage* the whole process is stopped and the user is informed to check the integrity of the datafiles on disk.
- After all instances of the application have finished the *write stage*, the elapsed times of all hosts is checked. Those hosts performing 10% [1] worse than the mean elapsed time are replaced by hosts that are expected to perform at least 10% better. On the other hand, if the host at the head of the free list is expected to perform at least 10% better it will be replace the slowest anyway.
- After the *write stage* the optional "**timesuspend**" is in effect. If the current time is within the period of time defined as *e.g. office hours*, all applications are stopped.
- On the other hand after completing the *write stage* the "**loadsuspend**" option may be effective. Using this option, the user specifies a maximum loadfactor for not-used hosts. Say, one has specified 0.3 as maximum. **Mcp** now checks the number of hosts that have exceeded a load of 1.3 during the past calculations. Just in case there are not enough hosts with a load below 0.3, all applications are stopped and execution is suspended until there is a valid number of hosts with a load below 0.3.

[1]In fact 10% is a tunable parameter called 'MARGE'

The *loadsuspend* and *timesuspend* options may be combined. Both privide adequate means to limit the use of the available machines to quiet hours or even quiet moments.

4 Options, parameters and interfacing

4.1 Application environment

The application environment can be defined as follows:

- The input channel (*stdin*) is used for commands only.
- The output channel (*stdout*) is used for replies to commands.
- The error message channel (*stderr*) is redirected to a file '.errors'. If this file existed, output on *stderr* is appended to the file.
- Each instance of the application is run in a unique directory. Directories have a two digit name, e.g. "rundir/03" for the third instance. The base directory ("rundir" in this example) can be defined by the programmer relative to the home directory. Global files can be accessed via relative paths: e.g. "../datafile.dat" is a global datafile.
- The HYDRA protocol consists of four letter messages with an end-of-line character at the end. When a programmer wants to test his/her application, the protocol messages can be entered manually as they are send and received via ordinary terminal equipment.
- To ease the use of the protocol, there is a hostlanguage interface for both C and Fortran.

The UNIX I/O channels *stdin*, *stdout* and *stderr* match the Fortran channels 5, 6 and 0 respectively.

4.2 The parameter file

Typically the parameter file containing all information to determine what should be done in the next cycle is local to the application, i.e. found in the current directory.

In general it contains two numbers:

- The number for this parallel instance. This number is set when initialising the directory structure and before starting the execution. It's a static number.

- The second number is the cycle number. It is incremented during the write phase. Initially it is set to 1.

Just is case you have local datafiles, which means that the work is split into parallel pieces by splitting data into pieces, the parallel applications need not know their parallel sequencenumber. In that case the first number may be omitted and the parameter file may be global to all applications.

4.3 HYDRA protocol

The table below gives an overview of the HYDRA protocol as used between the HYDRA's controlling process (**mcp**) and an application (**appl**).

message	symbol	sender	description
wait	M_WAIT	appl	appl. is started and ready
read	M_READ	mcp	appl. can read parameters a/o data
rdon	M_RDON	appl	appl. has completed the *read*
exit	M_EXIT	appl	appl. is finished
calc	M_CALC	mcp	appl. can start calculations
cdon	M_CDON	appl	appl. has finished calculations
writ	M_WRIT	mcp	appl. can write data to disk
wdon	M_WDON	appl	appl. has completed the *write*
trap	M_TRAP	appl	appl. has found an error
stop	M_STOP	mcp	appl. should stop

Some messages need more explanation. The *exit* message is given by the application when it has read the parameters and found that the next iteration to be done is not required. The *exit* message is sent instead of the *rdon*.

The *trap* message can be send to HYDRA at anytime when there has an error occurred. HYDRA in turn will restart the application at another host hoping that the fatal error does not occur overthere [2]. Fatal errors may include: cannot read diskfile, found unexpected things in the diskfile, couldn't get the amount of memory required. The programmer should write a clear error message to the *stderr* file for his/hers own benefit.

The *stop* messsage can be received at any moment, but usually after the *wdon* has been given. When HYDRA moves applications to other hosts the application to be moved is stopped after the cycle is completed and a new instance is invoked at a new host.

[2]There is an maximum number of retries on 'traps' defined. Currently: 10 times

4.4 Host language interfaces

For each host language there is one include file and two subroutines available. The include file contains the definitions of the symbolic names for the messages. A sample program in C looks like:

```
#include "mcpprotocol.h"

main()
{
    int message;

    sendm(M_WAIT);

    message = receivem();
    switch( message ) {
     case M_READ:
        ...
        break;
     case M_CALC:
        ...

    ...
}
```

The equivalent code in Fortran looks like:

```
#include "mcpfprot.h"
    integer mes
    call fsendm( M_WAIT )
    call frecvm( mes )
    if( mes .eq. M_READ ) then
        ...
    else if( mes .eq. M_CALC) then
        ...
    ...
```

The programmer needs to provide subroutines that perform the tasks as prescribed by the basic cycle. In most cases the routines performing the calculations can be taken from the original version with slight changes. Obviously, this depends on whether or not the original application was parameter directed.

4.5 Tunable parameters

The current version of HYDRA supports a number of tunable parameters contained in a parameter file called "mcpconf". Most of the parameters may be changed in runtime. This means that one may change "mcpconf" using a favorite editor (e.g. *'vi'*) and **mcp** will read it when the next package with systemloads is received from **rups**, or once a minute when **rups** is idle.

Parameter	sample value	may change	description
APPLPROG	"../progr"	no	executable application relative to current directory
APPLDIR	"rundir"	no	base directory relative to home
APPLNUMBER	8	no	number of parallel applications
RUNDOMAIN	"sci.kun.nl"	no	network domain of hosts
TIMESUSPEND	yes	yes	'yes' when timesuspend is desired
TIME_TO_STOP	"08:30"	yes	time to start office hours
TIME_TO_START	"17:30"	yes	time to restart applications
LOADSUSPEND	yes	yes	'yes' when loadsuspend is desired
MAXLOAD	0.2	yes	max load on not-used hosts
RUPSINTERVAL	30	no	number of seconds for rups to wait before polling the hosts again
MARGE	10	yes	% tolerance for not acting on slower hosts and ignoring faster ones
SIMULTANEOUS	yes	no	'yes' for simultaneous write stage is desired
NICELEVEL	5	yes	nicelevel for applications

RUNDOMAIN needs only be defined when mcp/rups run in a different network domain than the applications. Using this option, the hosts in the hostfile ("**mcphosts**") can be named without full domainname.

RUPSINTERVAL needs not to exceed the value of 45 seconds. In order to establish an accurate value of the mean load during the calculations **rups** should provide the table of loadfactors to **mcp** at least four to five times during the calculations.

4.6 Status display and error logging

HYDRA provides a display function to monitor the activity. The status display features a list of all applications with their status and assigned hosts with their load. In the lower part of the display, a list of free hosts is shown.

```
MCP v4 running 8 '../testapl'   cycle #4    MS_CALC  [ 32:28 ]

#  host       load                       status    remarks
-- ---------- -------------------------- --------- -------
01 groen       0.18 <                    AS_CALC
02 geel        0.83 >>>>                 AS_CALC
03 indigo      0.59 <<<                  AS_CDON   01:10 elapsed
04 oker        0.91 |||||                AS_CDON   01:11 elapsed
05 schwarz     1.08 >>>>>>               AS_CALC
06 bordeaux    0.73 <<<<                 AS_CDON   01:12 elapsed
07 cyaan       0.59 <<<                  AS_CALC
08 oranje      0.92 >>>>>                AS_CDON   01:19 elapsed

azuur    0.23/72       grijs    0.08/86        blanc    0.09/88
noir     0.05/93       wit      0.16/88        gris     0.13/91
gray     0.28/87       black    0.36/91
```

The display above shows a few applications that have finished. HYDRA is waiting for the others to finish. When all applications got the *c-done* state, the first gets the *write* command and then the second and so on. The display below shows the **write** stage; we're about halveway. Obviously, this is the one-by-one option of the **write** stage.

```
MCP v4 running 8 '../testapl'   cycle #5    MS_WRIT  [ 52:37 ]

#  host       load                       status    remarks
-- ---------- -------------------------- --------- -------
01 groen       0.35 <<                   AS_WDON   01:09 elapsed
02 geel        0.89 >>>>>                AS_WDON   01:17 elapsed
03 indigo      0.80 >>>>                 AS_WDON   01:12 elapsed
04 oker        0.92 >>>>>                AS_WRIT   01:10 elapsed
05 grijs       0.00                      AS_CDON   01:32 elapsed
06 bordeaux    0.88 >>>>>                AS_CDON   01:09 elapsed
07 cyaan       1.38 >>>>>>>>             AS_CDON   01:09 elapsed
08 oranje      0.79 >>>>                 AS_CDON   01:13 elapsed

azuur    0.21/72       gray     0.02/87        blanc    0.02/88
black    0.02/91       noir     0.02/93        schwarz  0.34/74
gris     0.31/91       wit      0.49/88
```

The headline of the display shows a.o. the cycle number, the overall state of **mcp** and the cummulative elapsed time. The histogram of the machineloads shows increase of load, decrease and equal loads compared to the previous load value. The list of free hosts shows their actual load and the *experience* value.

Mcp logs its events and errors to a file "Log.mcp", which looks like:

```
12/3 11:13:23 MCP started, pid=858.
12/3 11:13:23 load suspend enabled: maxload=0.50
12/3 11:13:23 Rups started pid=859
12/3 11:13:23 MCP tcp/ip portnr=1882
12/3 11:15:04 Execution resumed
12/3 11:15:04 start cycle 1
12/3 11:16:38 end cycle 1, 01:33 elapsed
...
12/3 11:19:03 start cycle 2
12/3 11:20:36 end cycle 2, 01:33 elapsed
12/3 11:20:56 Moved #2 from violet to geel
12/3 11:21:04 Moved #7 from gris to cyaan
12/3 11:21:04 start cycle 3
12/3 11:22:38 end cycle 3, 01:33 elapsed
12/3 11:22:39 start cycle 4
12/3 11:24:09 end cycle 4, 01:29 elapsed
12/3 11:24:32 Moved #5 from schwarz to grijs
12/3 11:24:33 start cycle 5
12/3 11:26:05 end cycle 5, 01:32 elapsed
....
12/3 11:40:59 start cycle 10
12/3 11:42:41 end cycle 10, 01:42 elapsed
12/3 11:42:48 Moved #8 from noir to azuur
12/3 11:42:49 MCP finished, pid=858
12/3 11:42:49 Total elapsed: 1:46:14
```

Applications are invoked by a program called "**acp**". On errors **acp** reports in a file called "acp.errors", which may be found in the home directory in case the working directory could not be reached.

The user's application report errors in a file ".errors" in their working directory. Each time error messages are appended to this file.

5 Implementation

5.1 MCP Internals

Mcp is an event driven application, that uses the *select* system call to find out which files, TCP/IP sockets in fact, contain readable input. Internal bookkeeping provides the link between activity on a file on one hand and status change for an application process on the other hand.

Mcp has an inner loop that can be described as follows.

1. For each connection with an application and for the connection with **rups**, a bit is set in a *bitmask* for *select*.
2. The *select* system call checks the files marked in the bitmask for input. When *select* returns at least one has input for **mcp**.
3. The bitmask returned by *select* is checked to see what files have input. The channel to **rups** is checked first. If **rups** has sent a package of loadfactors the status of all hosts is updated and the freelist is reordered. Acting on hosts that presumably went down is done here and now. After **rups** has been handled all files associated with applications are handled if they have input for **mcp**.
4. Each application that has input for **mcp** is handled: the message is read and a status change is booked in the status table. Each application has a previous and a current state and above all **mcp** has a general state.
5. Each status change results in a loop in the two-level stage machine. The state machine consists of a state table defining which routine to call in a certain situation. The state table is searched by current **mcp** state, current (new) application state and previous application state. Each entry in the state table *may define* a routine to be called on the applications state change and a routine to be called when all applications have reached a certain state specified in the entry. This last item is used to specify actions to be done when all applications are in a equal state and work may continue with another *stage* or *cycle*.
6. Subroutines called from the state machine implement detailed actions to be taken in circumstances defined in the state table.
7. When all active files are handled the loop returns to step 1.

A mayor design consideration is to minimize the need of timeouts. With timeouts on all asynchronous actions the complexity would exceed the current straight forward implementation.

This implies that when a user's application hangs in any stage the whole thing will wait until user interaction takes place. The only important step that is garded with a timeout is the remote startup of an application proces.

5.2 Load measuring and performance estimates

Standard Unix programs, such as *uptime* and *rup* provide indications on the load of a machine. The *loadfactor*, mentioned earlier, is defined as the mean number of processes being served by the CPU in the past period of time. In theory this means that when three processes obtain the CPU the loadfactor is 3.0. In practice the loadfactor may be less, partly caused by system overhead and the interruptions of very small length: i.e. someone typing to an editor.

In HYDRA it's of importance to value available hosts not only by their current load, but also by their relative performance. Hosts with a lower load may perform worse than a more powerful host with a bit higher load.

When a host in the hostlist has been used at least one time, there is a so called *experience* value defined:

$$Experience = \begin{cases} \frac{elapsed\_time}{\overline{load_{calc}}} & if\ \overline{load_{calc}} > 1 \\ elapsed\_time & \text{otherwise} \end{cases} \tag{1}$$

The order of hosts in the free list is according to expected performance. For hosts that have a defined *experience* the expected performance is:

$$expected\_performance = (load_{actual} + 1.0) \cdot Experience \tag{2}$$

Hosts that have not been used yet, have an undefined *experience* value and are sorted by their actual load and come preceding hosts that have been used before.

Both the calculations on *experience* value and the ordering by *expected performance* assume the load caused by the user's application is 1.0.

This assumption only yields when the computing process does not read data from disk and is computing all the time on a machine with low system overhead. System overhead increases dramatically when more processes are active or the network is used heavily. On the other hand the assumption is applied to load factors measured by **rups**. Rups only collects the load every now and then, asynchronous to the status of the application. During calculate stage the first measure of load is dropped and only the rest of the measures is used for calculating the average load during calculate stage.

This way of dealing with the ordering in the free list of hosts and the calculations for *experience* and *expected performance* privides a suitable way to distinguish machines of different power.

5.3 Starting scenario

The startup scenario for a new application on a remote machine is synchronous and blocks all activity until completed. There is a single timeout on starting a new application. To start an application on a remote machine the following scenario is used.

- **Mcp** starts *acp* on the remote machine using *rsh*. It provides the name of the working directory, the nice level and the pathname of the user's application as parameters to *acp*.

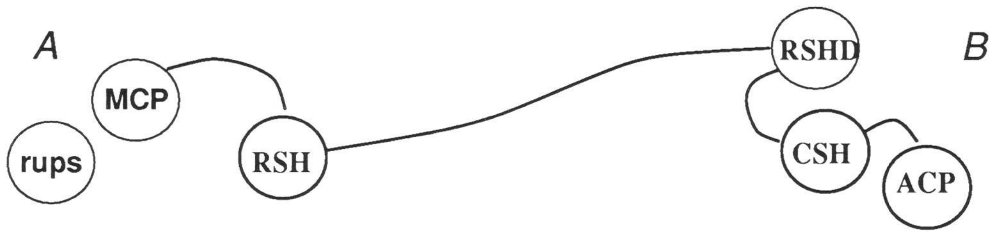

Figure 6: acp started remotely by rsh

- **Acp** forks itself into a second instance. The first instance quits, which implies that the communication channels built by *rsh* are removed. As a result we have *acp* running on the remote machine with no connection at all to *mcp*.

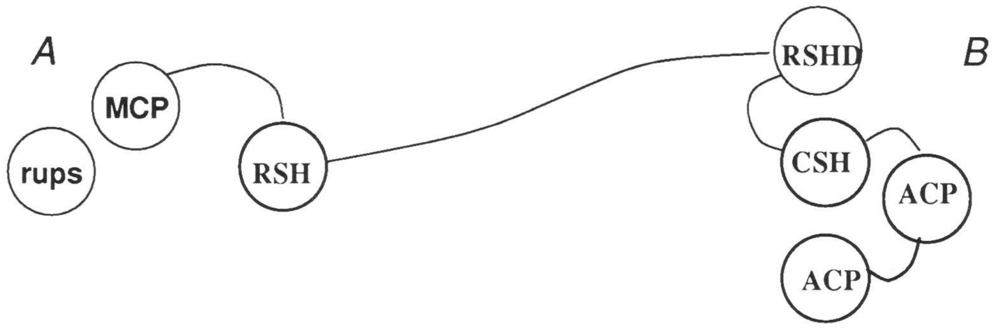

Figure 7: acp forking itself

- **Acp** (the second instance) moves to the desired working directory, initiates a *TCP/IP socket connection* to **mcp**, sets the desired nice level and redirects *stderr* to append to the file ".errors".

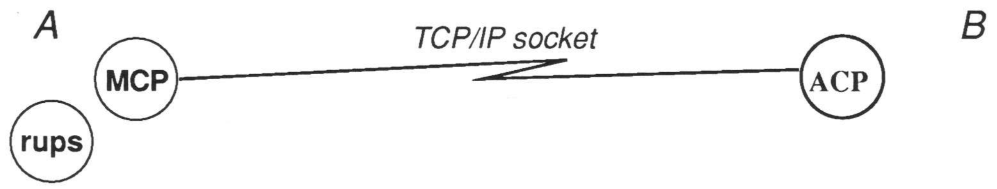

Figure 8: TCP/IP sockets built between acp and mcp

- **Acp** *execs* the application. The application enherits all the environment and open files (including the TCP/IP socket).

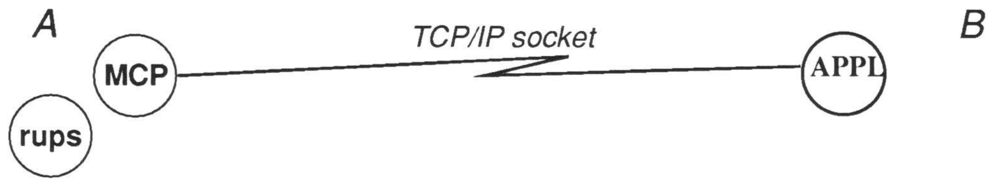

Figure 9: acp replaced by application

As a result the application runs on the remote machine with a communication channel to **mcp**. This channel is being used by the application as if it were the terminal display and keyboard [3].

6 Conclusions and remarks

The HYDRA package provides a flexible means to control long running applications in a networked environment. Using the parallelism is restricted to a certain class of applications. The contraints for the parallellism effect independency and similarity of the *steps* of a total application.

HYDRA has first been used to run a Neural Network application during six weeks on 12 machines in parallel. The total time on one single machine is estimated over 450 days.

[3]This is the main reason why applications can be tested without using HYDRA: their commands use *stdin* and *stdout*.