Automatic Operations
for the CRAY XMP48
under Unicos

Vincent Doré\(^1\), and Achille Petilli\(^2\)

Abstract

The Cray XMP48 running in the CERN computer center offers, as all other CERN’s mainframes, a 24h a day, 7 days a week service. Our computer center is permanently manned.

Due to the increasing complexity of the operating systems, the amount of human interventions needed by any system will have to decrease. This paper describes the first steps we are taking to ensure smooth operations of our Cray, while reducing the need for human operators and still providing the same level of service to our users.

\(^1\) CERN, CN Division, Computer Operation
\(^2\) CERN, AS Division, Management Information
1. Introduction

In the last few years computer systems, have become more and more complex and powerful with respect to both the hardware and software. Users have more and more facilities available. The same is not completely true for the operational staff.

While hardware reliability has been steadily increasing, software reliability has not followed, mainly because of increased complexity. For instance on our Cray system, roughly 2/3 of interruptions experienced the last two months are due to software while hardware errors account for the rest.

Nowadays operators find themselves confronted daily with obsolete documentation, incomplete procedures, unexplained system behaviour, etc. A much more complete knowledge is required to effectively run a machine; the whole job of an operator is basically rapidly changing.

We are now going to present in some more detail the problems we want to solve and the way we believe will be the most effective one to do so.

1.1 Goals

Our main goal is to keep our Cray in a proper, working state with reduced manpower. Eventually this manpower could be totally unavailable for some extended period of time (e.g. nights, week-ends). In making this effort we want also to provide a service at least as good as the current one.

Even though the goals, and the reasons behind them, are easy to understand, the way to achieve them is not completely clear. We started our work with a few first cut design decisions that we felt primordial to provide a working system.

1.2 Basic Ideas

We feel that the changes introduced by an Automatic Operations tool are drastic enough to be potentially disruptive for the services provided. For this reason a first decision was taken: keep it simple, without for this compromising our objectives.

A second decision was dictated by our experience with Unix\(^1\) (and other operating systems as well) and on the consideration that many problems will disappear, reappear and/or change, their visible effects depending on the software release and on the local configuration/usage. The whole system should be modular enough to allow for prompt modifications while minimizing side effects.

We realize that the maintenance effort has killed many projects and our own is going to be no exception. To prevent such a failure we want to be able to simulate and test the running system without affecting normal operations. This leads to a strong emphasis on documentation and test programs.

Before we enter into details, let us clarify a little bit the sequence of actions an automatic operator would have to follow.

---
\(^1\) Unix is a trademark of AT&T.
We want to be able to collect information about the state of the machine and detect worthwhile\textsuperscript{2} events that can be processed in a standardized way. Events will then be analyzed and they will trigger some related action. Every action will produce results aimed at correcting the situation that started this sequence. In the case of a human operator, there is also a process of feedback, that provides for additional actions to be taken should the previous ones be ineffective.

In what follows there is a more detailed description of the implications of our decisions.

1.2.1 Keep it simple

This does not mean do only simple things. Rather it should be taken as do everything in a logical, coherent, well understood way. We believe that the sequence of operations depicted above is well in this spirit. It also closely matches the way a human being would attack the problem. Keep it simple also means make use of existing programs/tools to solve our problems.

1.2.2 Modularization

There is no need for any of the above steps to know about the others: to effectively decouple each one of them we should standardize on an interface between the various steps. This way we can also keep each module separate and then replace whichever one needs to be updated without affecting in any way all others.

1.2.3 Documentation and Tests

Keeping all different parts of the system separate also produces another result: documentation on the interface between the different parts will be all that is needed to understand the system, to test it and to maintain all the different parts of it. Writing test programs is also made easier.

1.3 Design Conclusions

With the ideas we have just described, we became confident that the whole concept of automatic operations is a workable one. Using simple, already existing and documented tools (e.g. Unix shells) we can even take advantage of operators' experience to help us build up and run such a system. This also reduces the unavoidable resistance to changes that the introduction of an automatic operator will surely introduce amongst the shift staff.

\textsuperscript{2} Worthwhile from the point of view of the operations.
2. Technical Design Considerations

We are now going to explore in detail the technicality of our automatic operator implementation.

2.1 Survey of the different tasks

The inner working of the automatic operator on the CRAY relies on five different levels. They are:

1. the collection of the various statuses and error messages which are provided by the computer in different ways,

2. the parsing of the collected messages and the attribution, for each of them, of a problem code,

3. the analysis of each problem number and the choice of the action(s) to activate,

4. the action(s) to execute as a result of each decision,

5. the analysis of the past actions performed and their results.

The different parts of the automatic operator and their relation are symbolized in picture 1.

In the following there is a start analysis of each of the previous steps.

2.2 Collection of the messages provided by our CRAY

Unicos running on the CRAY XMP48 supplies error and status messages in different ways. Some of these messages go directly to the main console, others are sent to different log files. These various messages will be collected at the different places and then sent to the parsing process. In addition to the messages spontaneously provided by the operating system, different checks will be performed regularly (i.e.: the load of the machine, the proper working of the hardware, etc). The results of these checks will be sent to the parsing process.

2.3 Parsing of the messages

The goal of the parsing process is to compare the messages provided with a list of standard messages of which the automatic operator is aware. It will thus be able to choose for each message a problem code. It will send this code, with its associated priority code and possibly with some additional information, to the decision process. The parsing process has been written in an easy to update form, since the messages and the problems are subject to change in the future versions of the operating system.
2.4 Decision process

The decision process decides what to do for each problem code it receives. In the case that several problems have to be processed simultaneously, the associated priority code will help determining the first one with which to proceed. The decision is taken in accordance with the previously received problem codes and the global system state. For each received problem code, the decision process launches one or more new process(es), each of them running specific action(s). Problem codes and their respective actions are coupled by their names. Thus the update of the system is possible without modifying and recompiling the decision process.

2.5 Actions

The actions to perform are dependent on the problem discovered. Typically, for each problem code, the decision process can:

1. write a message to the oper communication display and possibly wait for an answer,
2. start a specific action, check its execution, and send back the result to the parsing process,
3. write in a logfile the action it has tried to carry out and its result,

4. send a mail to the system administrator.

One, several or all of the above steps can be taken, depending of the nature of the problem.

2.6 Analysis of the past actions

This process uses the logfile previously created by the decision process. It regularly checks the last actions taken and their results. It is thus able to detect problems which cannot easily be noticed by the decision process (e.g. recurring problems). For each new problem detected, a problem code is sent to the decision process which then acts as described above.

3. Technical decisions

3.1 Generalities about the processes

Two kinds of processes are used for the automated operations. The first ones (collection, parsing, decision and analysing processes) are daemons running as long as the automated operations are on. They are launched, monitored and stopped by specific shell scripts. The second ones are triggered processes. These are started by the decision process and stay alive the time needed to perform their goal. According to the actions to perform they can be either C programs, shell scripts or anything else adapted to the requirement.

As mentioned above, having the processes tightly bundled together would be a handicap in our framework. We tried to build each process to be as independent as possible from the others and we rely on interprocess communication to make them cooperate.

3.2 Interprocess communications

Under Unix System V, there are several tools that allow processes to communicate. They all rely on some common storage space. In order to find the best solution, we shall examine for each considered tool:

1. how to manage the common space,

2. how to manage concurrent write and read accesses to the common space,

3. how to manage several processes sending data to the same reading process,

4. to what extent can two communicating processes be independent.

By independent, we mean that it should be possible to stop and restart one of the communicating processes without pertubing the other one and without loosing any data. This leads to the concept of an identifiable communication channel where either the writing or the reading programm can be individually (re)connected.
3.2.1 Using pipes

A pipe is the main Unix way to make two processes communicate. It connects the output of a program to the input of a second one.

The main advantage of the pipe is the simplicity in writing the cooperating programs (there is no need to explicitly manage the common buffer space, neither concurrent read and write access). But,

1. several programs sending data through the same pipe need an additional mechanism (as semaphores) to avoid data intermixed into the pipe causing unexpected results,

2. if using unnamed pipes, sending and receiving programs are tightly bundled together (i.e. they must descend from a common ancestor process).

3.2.2 Using files

Temporary files are another possible solution. Programs write data into files that are processed by a reading program. Processes are independent enough. In case of failure of the reading program, data is still in the temporary files where it stays until read.

Care has to be taken when managing the common space and getting rid of the processed data. To avoid sequential files always growing and difficult to shorten when the programs run, files should be organised as a circular buffer. Such files need a complementary mechanism to synchronize cooperating programs. They have to be big enough to keep all the unprocessed data safely. Cooperating processes are totally unrelated.

3.2.3 Using shared memory

Shared memory can be used as a mailbox to allow programs to communicate. The access to the shared memory has to be controlled to prevent its access by two different programs at the same time. As files, the shared memory should be organized as a circular buffer. Processes are totally independent. In case of failure of the reading program, data can be stored until it can be processed.

3.2.4 Using a message queue

A message queue allows programs to send data to receiving programs with the following advantages:

1. the cooperating programs do not have to worry about managing common space, neither its access,

2. several programs can write without difficulty through the same queue,

3. cooperating programs can be run and stopped independently, without loss of data.

3.2.5 Using semaphores

Strictly speaking, semaphores are not a way of sharing data between processes. They allow the sharing of flags and are useful to manage cooperating processes. They can be used as a mechanism to control concurrent access to a shared memory segment or a pipe.
3.2.6 Summary

As shown above, message queues are the simplest tool which would fit our needs. Unfortunately, Unicos does not provide such tools at the present time (the msgget(2), msgctl(2), msgop(2) system calls are not a mandatory of Unix System V). Neither does it provides shared memory or semaphores.

The choice is thus restricted between files and pipes. Using files has the advantage of a good independence between sending and receiving processes. But, as it decreases efficiency and involves extra work in managing the files themselves, we decided to use named pipes.

3.3 Building our own communication tools

Our communications tools are some simple functions which provide features similar to the message queue calls. The functions rely on a named pipe (fifo) allowing several processes to send data to a unique receiving process. The access to the pipe by the writing programs is managed by a lock on an auxiliary regular file.

Six functions manage the communication between processes:

```c
int opensmsg(files, fd)
char *files;
int fd[2];

int sendmsg(fd, message)
int fd[2];
char *message;

int openrmsg(files, fd)
char *files;
int fd[2];

int readmsg(fd, message)
int fd[2];
char **message;

void freemsg(message)
char *message;

int closemsg(fd)
int fd[2];
```

Opensmsg() opens a communication channel for writing; the process can then send messages to another one. The first argument is the common file name prefix used for the communication files. The named pipe name is files concatenated with _e, the access control file name is files concatenated with _c. The second argument is an array of two integers: the file descriptors of the opened files.

Sendmsg() will send a message through the pipe. The first argument is the array of file descriptors obtained by a previous call to opensmsg(). The second a pointer to the message to send.
Openmsg() will allow a process to read messages coming from others. The arguments are identical to those used in opensmsg().

Readmsg() opens a communication channel for reading; the process can then receive messages. The first argument is the array of file descriptors obtained by a previous call to openmsg(). The second is the address of a pointer to an array of characters. After a successful call this pointer will point to the received message. As the space needed to store the message is obtained by a call to malloc(3C), it will be necessary to deallocate it when no longer needed.

Freemsg() will deallocate the space used by a received message.

Closemsg() will close the files opened by opensmsg() or openmsg().

As they are implemented, these functions have some limitation compared with the standard message queue calls:

1. only one process will be able to read the messages coming from the others, but this is sufficient for our needs,
2. they do not provide the various control options provided by msgctl(2) which are superfluous to us,
3. they do not classify the messages according to their type, thus we will have to use several pipes to read messages according to their priority.

On the other hand, readmsg() dynamically allocates space for the received message. Thus we do not depend on a fixed length of message as implemented in msg.h(3).

3.4 Communication between processes

3.4.1 Between the information gathering processes and the parsing process

A message queue is a suitable way for communicating between the various collection processes and the parsing process. We use the functions described above. All the collected data is sent to the same pipe where it is read in a fifo (first in first out) order by the parsing process.

3.4.2 Between the parsing process and the decision process

There is no way for the decision process using the functions described above to select messages in a pipe according to their priority code. For the time being, we make no use of priority codes. To cope with a priority scheme we will have to set as many pipes as priority codes.

3.4.3 Between the decision process and the action

For each triggered action, the decision process triggers a new process. Communication between the decision process and the new one are done through a pipe. A simple function popenr() (in the style of popen(3S)) creates the pipe and manages bidirectional communication.
3.4.4 Between the decision and the feedback processes

The feedback process looks for information in logfiles written by the decision and actions processes. It then sends back information to the decision process just like another gathering process.

3.5 Processes specification

3.5.1 The collecting processes

Without any priority scheme, collecting processes use the same message queue to send data to the parsing process. Nevertheless, as an implicit synchronisation is done by the communication functions, none of the collecting processes has to worry about any of the others. Thus changes in one of them, adding or suppressing one of them, does not require modification to the others.

3.5.2 The parsing process

3.5.2.1 Requirements

Special care has to be taken in designing the parsing scheme. We do not want a monolithic daemon which is difficult to update, as messages can change each time a new version of the operating system is provided. To avoid long and tedious update work, the parsing program should:

1. be able to detect new messages,
2. allow an easy update of the known messages.

In the following, we examine the possible solutions to the above requirements.

3.5.2.2 Using awk

`Awk(1)` is a text processing language which is designed to perform text analysis and manipulation. It is simple to learn and use. Books presenting its features and usage are also available.

Messages arriving to the decision process can be analysed by awk. The decision process can invoke awk through a pipe for each new message. Awk directives to analyse the messages would be saved in an independent text file which can be simply checked and modified. Thus syntax changes, addition or suppression of text in the system messages would not require any modification in the parsing program itself.

According to the instructions specified in the directive file, awk will send back to the parsing process a problem code, a priority code and optional additional information. The three of them will immediately be sent to the decision process.

3.5.2.3 Using C code

There are several ways of writing the parsing process with C. All the work can be done in the C program itself. This implies modification and recompilation of the program each time the system mes-
sages change or there is a different interpretation of a message. In our context, having to deal with fre-
quent changes, this is not a very good solution.

Another solution can be to maintain outside the C program both a table of messages and a set of
directives to process them. We need to create a new tool (directives and the way to use them) from
scratch. Setting up a new tool doing more or less the same as other existing ones (awk, sed, etc...)would involve a considerable amount of extra work and added debugging and maintenance effort.

3.5.3 Using Lex et Yacc

Lex an Yacc could effectively be used to design the parsing process. Nevertheless they are too
sophisticated for our needs, and they require knowledge of them and C programming ability to be able
to change the parsing work.

3.5.3.1 Conclusion

Awk is reliable, easy to use and well documented. Using awk, the analysis of the system messages
could be written by anybody who has a minimal knowledge of programming. Changes can be tested
interactively and do not need any recompiling. Furthermore, changes can be put in production even
without stopping the automatic operator.

3.5.4 The decision process

The decision process will initiate two distinct kinds of actions:

1. general actions, which act upon every incoming message (logging of the event),
2. specific actions which will depend on the content of the incoming message.

The first part of the decision process treats the general actions. The second generates (after the
relevant checks) a new process that launches the specific action. The specific action could be selected
through a big case statement handling each problem code. More efficient is the use of specific actions
whose names are directly related to the associated problem code. This allows treatment of new incom-
ing messages without any modification and recompilation of the decision process.

For each received code, the decision process finds, in a dedicated directory, the list of actions to
perform. The names of the actions to perform are the name of the problem code followed by a num-
ber which will determine the order in which the actions are launched. Each action started will send
back data which will be written in a log file and a status code. In case of a non zero returned status,
the decision process will not start the following actions.

The additional information, optionally coming with a problem number, will be passed, without
any change, as an argument to the specific action(s).

There is no special requirement on how to write the specific actions. Shell scripts can be used as
well as compiled programs according to the action to take.

We should also consider the possibility of sending messages to the operator console, possibly
waiting for an answer. Some problem to be treated by the automatic functions will need to notify the
operators, other not. To be as flexible as possible it seems correct to let the outside specific action
cope with such messages, even if it leads to some redundancy. Another solution could have been to add specific information coming from the parsing process to signal the decision process it has to notify the operators. But this complicates too much both the awk file and the decision process and is not very flexible.

3.5.5 The feedback process

The feedback process is the most sophisticated one. It has to be intelligent enough to get a global view of the recent events and to deduce what and if extra action is needed to keep the system in a working state. This is an addition to the above described automatic operations and will be set as the last part of the system, after all the other steps are working.

It will basically get information from the logfile messages being written by the decision process and the specific actions. When required it will communicate with the decision process, through the message queue also used by the information gathering processes, using its own problem-number set.

3.6 Foreseen maintenance and work

According to the above described scheme, we can divide the maintenance work in four parts related to:

1. the parsing and the decision daemons,
2. the collection daemons,
3. the feedback daemon,
4. the awk file and the specific actions.

Writing the daemons is work of a dedicated programmer. After being written and tested, the parsing and the decision daemons should not require any maintenance work, apart from recompiling and testing every time a new system is installed. The collection daemons will require some changes each time there is a need to modify the collection of data (typically new daemons will be added to check some additional problems, some others will be disabled). The feedback daemon will be less easy to maintain and its final tuning could take some time.

Maintaining the awk file and writing (or modifying) specific actions will be done by whoever is involved and who has knowledge of shell programming and awk manipulation. Typically for each message collected the work will be:

1. to write the matching pattern awk statement,
2. to write the corresponding awk action, sending back to the daemon a problem code, a priority code, and some optional information,
3. to write the specific action(s) associated with the problem number, using the optional information selected by the awk action.

These “outside” awk file and specific actions should keep the maintenance work reasonable. After an update of the operating system (if this does not create new unexpected problems), it should be sufficient to adapt the awk file to match the new syntax of the messages, if necessary.
3.7 Implementation schedule

1. writing of the interprocess communication functions,
2. writing of the parsing process (without priority scheme),
3. writing of the decision process,
4. writing of the first collection processes (handling disk space limits and tape errors) and their corresponding actions,
5. tests of the system,
6. addition of collecting processes and corresponding actions,
7. addition of the priority scheme,
8. automatic transfer and analysis of the logfile on a workstation,
9. implementation of the feedback process,
10. tuning and refinements.

The first three steps are fully done as of today, part of step #4 is implemented (disk space problems) and step #5 has been done with respect to the existing parts of the system.

4. Conclusions

We reviewed carefully all interventions our operators did in the last months on Unicos. The most frequently recurring problems, in order of decreasing frequency, are:

1. Disk space,
2. Tape unit failures,
3. Disk flaws,
4. Degraded service (i. e. CPU failure resulting in one CPU being turned off),
5. Backup failures.

Of course there are many other problem areas, but they are either outside the competence of the operators or not frequent enough to justify our attention for the time being.

The first prototype system has been operational since mid–November 1989, with some of the information collection processes ready and action procedures consisting simply of a message sending program (msgr(1)/msgi(1)).

A full-fledged production system is planned for mid 1990, with at least the three frequent most problems being handled.
Future development will include the extension of the system to the Sun systems that act as the Ethernet gateway for the Cray.

5. First results

We have installed a first version which reports successfully each disk space threshold encountered. In the immediate future, we plan to integrate in the automatic functions the program which tests and possibly disables CPUs of our CRAY as both the detection process and the related action already exist. Tape unit monitoring will be introduced next.