Computer Maintenance via Batch Execution

Martin MAREŠ[1]
Department of Applied Mathematics
Faculty of Mathematics and Physics
Charles University in Prague
Malostranské nám. 25
118 00 Praha 1
Czech Republic
e-mail: mares@kam.mff.cuni.cz

Abstract

Organization of programming contests often involves maintenance of hundreds of computers. While parallel installation of operating systems is a well understood problem, further maintenance of installed systems is often cumbersome. We present a simple, yet powerful batch processing system, which makes this task easier.

Keywords. cluster maintenance, batch processing

1 Introduction

Large programming contests, like the International Olympiad in Informatics, ACM ICPC, or many regional competitions often involve hundreds of computers. Some machines are used directly by the contestants, while the others work behind the scenes as contest servers or worker nodes of a grading system.[2]

The whole infrastructure is usually set up for the duration of a single contest. Within a couple of days, the computers need to be installed and configured for their intended roles. During the contest, further administrative actions have to be performed: distribution of files related to contest tasks, fixing of minor bugs, pro-active checking for hardware problems, and so on.

Generally speaking, administration of the contest network is similar to that of a computer cluster. We will follow the terminology commonly used for clusters and call an individual computer a *node* of the system. Unlike a typical cluster, our nodes can act in different *roles,* so it will be useful to divide them to *groups* such that all nodes within a group are handled in the same way.

Parallel installation of software to multiple nodes, including the operating system, is a well understood task. The nodes can be booted via network (e.g., using PXELinux by Anvin et al. (2013)) and execute an automatic installation script. Then the software can be installed package by package, for example by a tool like FAI (Lange et al. (2013)). Alternatively, a full disk image can be copied. The latter approach is less flexible, but it can be significantly faster if a multicast-based or tree-based distribution protocol is used (see Knaff et al. (2012) or Mareš et al. (2009)).

[1]Partially supported by the Center of Excellence – Institute for Theoretical Computer Science, Prague (project P202/12/G061 of the Grant Agency of Czech Republic).

[2]For a description of a typical contest system, please see Mareš (2009) or Maggiolo and Mascellani (2012).

Management of already installed nodes is a more complex problem. A part of its complexity lies in the ad-hoc nature of maintenance tasks. It is therefore hard to find a general framework where all such tasks fit. On the other hand, most tasks can be expressed as shell commands. There are multiple tools for running shell commands in parallel on a group of nodes. A typical example is ClusterSSH by Fergusson (2010): it opens one terminal window per node, the user then types commands in a master window and the keystrokes are broadcast to the other windows.

ClusterSSH is easy to use, but it has several drawbacks. First, it is hard to express tasks, which need to be applied conditionally, depending on the state of the node. More importantly, it fails when a node is inaccessible due to a temporary hardware or software problem. With an increasing number of nodes, the probability that there is a faulty node converges to 1. We need to save the failed commands and repeat them later, when the node recovers.

These problems have led to development of complex configuration engines – most notably GNU cfengine (Burgess (1995)) and Puppet (Puppet Labs (2013)). They are given a declarative description of the intended state of the cluster (which packages should be installed where, which user account should exist, etc.). The engine then periodically checks if the nodes conform to the description and attempts to fix any discrepancies.

This approach is robust with respect to software and hardware failures. It also makes repeated administrative tasks (e.g., installation of software packages) easy, as long as a parametrizable template can be written. On the other hand, one-of-a-kind tasks take too much effort to declare and the complexity of the engine makes failures unnecessarily hard to diagnose.

We have designed and implemented a new cluster management tool, built upon queues of batch jobs. It retains the straightforward simplicity of shell-based tools, but hopefully eliminates most of their drawbacks.

In the following sections, we describe the data model of our system. Then we discuss operations defined on the model and their reference implementation.

2 Data model

The data model used by our system consists of multiple instances of two simple building blocks: jobs and queues.

2.1 Jobs

A *job* is an abstraction of an action, which can be executed on a node. Each job has a header and a body.

The *body* of the job is a script to be executed – by default, it is a shell script, but other interpreters, like Perl or Python, can be used.

The *header* is a collection of *job attributes,* written as arbitrary key/value pairs. The attributes can influence handling of the job. The basic attributes include:

- *ID* – a unique alphanumeric identifier of the job. If unspecified, the identifier is generated automatically, but jobs can be also given meaningful names.

- *Subject* – a one-line textual description of the job, intended to make lists of jobs comprehensible.

Execution of a job on a node consists of three phases:

- *Preparatory phase:* the body of the job is transferred to the node, possibly wrapped in a common *prolog* and *epilog* code (outside obvious use for initialization and cleanup, the prolog can also serve as a library of functions available to all jobs).
- *Running phase:* The script is run on the node, either within a terminal or non-interactively, depending on system configuration.
- *Cleanup phase:* The files related to the job are removed from the node.

Additionally, each job can request special handling in the preparation and cleanup phases. The attributes of the job can include shell commands to be run in these phases. The job can also specify its *attachments* – extra files uploaded to the node in the preparatory phase, available to the job during its execution in its current working directory, and removed in the cleanup phase.

Let us consider possible failure modes. A job can fail in the preparatory phase, in which case we just abort its execution (and optionally run the cleanup phase to remove all traces of the job). If a cleanup fails, the remnants of the job have to be removed manually (or by retrying the cleanup later). When running the job fails, we cannot be sure which part of the job was completed; it is even possible that the whole job was run, but the connection broke before the information about completion was transmitted. In all such cases, the execution of the job can be retried later. Because of that, all jobs are generally expected to be idempotent.

2.2 Queues

Scheduling of jobs on nodes is represented by *queues.* Each queue keeps its own set of jobs and known nodes. For each node, some of the jobs can be enqueued (scheduled for execution).

The jobs can be assigned to nodes individually, but it is more common to enqueue a job on a *group* of nodes. Groups are defined in the configuration of the system; each group contains a set of nodes and possibly includes other groups. When a job is enqueued on a group, the group is immediately expanded to its constituent nodes.

Later, a queued job is executed. The result of the execution is recorded in a *status file.* If the execution was successful, the job and its status file are moved to an archive of historic jobs. Otherwise, the job is kept in the queue and its status reported.

For each host, the queued jobs are naturally ordered by their identifiers and usually executed in this order. As the automatically generated IDs are based on timestamps, the jobs are by default executed from the oldest to the newest. However, applications requiring specific order can issue specific IDs.

Generally, all jobs in the queue can be executed simultaneously, but this is seldom welcome. The queue therefore specifies a *locking model,* which restricts the degree of concurrency. The default model forbids multiple parallel jobs on the same node, but parallelism between nodes is allowed.

3 Implementation

We have written a reference implementation of our queueing system. It was developed on Linux and it should run on all POSIX-compliant operating systems. It is available from `http://mj.ucw.cz/sw/bex/` and it can be distributed under the terms of the GNU General Public License.

The implementation is written in the Perl language in a spirit similar to the Git revision control system (Hamano and Torvalds (2013)). It consists of a core library, a driver program `bex`, which parses arguments and passes control to different subcommands, each handling one type of operations.

The whole program runs on one node, designated as the master of the cluster. It communicates with the other nodes via the Secure Shell protocol (SSH), leaving all the intricacies of authentication and encryption to it. No special software is required on the other nodes, except for basic system tools.

The following basic operations are supported:

- `bex qman` – manage queues: list available queues, or add a new one.
- `bex add` – add a new job and enqueue it on a set of nodes and/or groups. The job can be created either interactively in a text editor, or completely specified by command-line options. It is also possible to take a job which was already completed and re-queue it for repeated execution.
- `bex queue` – inspect a queue. Shows jobs in the queue, together with their status. The listing can be filtered to show only specific jobs, specific nodes or for example jobs whose execution failed. Additionally, jobs can be removed or moved to a different queue.
- `bex run` – run jobs sequentially. Takes a queue and runs all queued jobs one after another. Again, the set of jobs can be limited to a subset of nodes, or a specific job. Job locking rules are respected, so it is safe to run multiple instances of `bex run` simultaneously.
- `bex prun` – run jobs in parallel. Takes a queue and runs all queued jobs, as many at a time as allowed by the configuration. Locking rules are again respected (in fact, `bex run` is run internally to handle the low-level details). Current status of all nodes and jobs is shown on a status screen.

All jobs are allowed to be interactive, i.e., require a controlling terminal. When we run jobs sequentially, they inherit the terminal from which `bex run` was started, tunnelled via SSH. Parallel execution employs a terminal multiplexer to create a virtual terminal for every running job. An obvious choice is GNU Screen (Chowdhury et al. (2013)), but we were unable to create new virtual terminals on the background, which made interaction with the parallel execution almost impossible. We therefore use `tmux` (Marriott et al. (2013)) instead, whose design is less convoluted and which has much better remote control capabilities.

4 Conclusion

We have designed and implemented a new cluster management tool, based on parallel batch scheduling. The tool is very simple, but several years of experience with its use have confirmed that it is very effective. We have been using it to manage a network of about 60 computers at our department and also to manage contest systems at several programming competitions we organize.

The implementation should scale well to hundreds of nodes. The only problem we are aware of is the layout of the status screen in `bex prun`, which may become hard to read; this however does not impact running of the jobs and furthermore it should be easy to fix.

With an increasing number of nodes, SSH connections could become a bottleneck. Experimental results show that this is not an issue as long as the jobs are reasonably small. If huge attachments are needed, the performance can be limited not only by SSH, but also by the bandwidth of the network interface of the master node. Such cases can be worked around by using a clustered filesystem to distribute the files. A more systematic fix, which should not be hard to implement, could be to allow multiple instances of `bex prun` running on different master nodes and sharing a queue.

In the future, it could be useful to replace ID-based ordering of jobs by a full system of dependencies between jobs. This would allow parallel execution of independent jobs on one node, or operations like "requeue a job together with all dependent jobs". It would also lead to better handling of errors.

References

H. Peter Anvin et al. (2013). The Syslinux Project. `http://www.syslinux.org/`.

Mark Burgess (1995). A Site Configuration Engine. USENIX Computing systems, Vol. 8, No. 3. Available from `http://www.iu.hio.no/cfengine/`.

Sadrul Habib Chowdhury et al. (2013). GNU Screen. `http://www.gnu.org/software/screen/`.

Duncan Ferguson (2010). ClusterSSH. `http://clusterssh.sourceforge.net/`.

Junio Hamano, Linus Torvalds et al. (2013). Git Revision Control System. `http://www.git-scm.com/`.

Alain Knaff et al. (2012). Udpcast. `http://www.udpcast.linux.lu/`.

Thomas Lange et al. (2013). FAI – Fully Automatic Installation. `http://fai-project.org/`.

Stefano Maggiolo, Giovanni Mascellani (2012). Introducing CMS: A Contest Management System. Olympiads in Informatics, Vol. 6, 86–99, Vilnius, Lithuania.

Martin Mareš (2009). Moe — Design of a Modular Grading System. Olympiads in Informatics, Vol. 3, 60–66, Vilnius, Lithuania.

Martin Mareš et al. (2009). The shcp tool, part of the Sherlock Holmes Search Engine. `http://www.ucw.cz/holmes/`.

Puppet Labs (2013). Puppet Open Source. `https://puppetlabs.com/puppet/puppet-open-source/`.

Nicholas Marriott et al. (2013). TMUX. `http://tmux.sourceforge.net/`.

M. Mareš is an assistant professor at the Department of Applied Mathematics of Faculty of Mathematics and Physics of the Charles University in Prague, organizer of several Czech programming contests, member of the IOI Scientific Committee, and a Linux hacker.