GenWrapper: A Generic Wrapper for Running Legacy Applications on Desktop Grids

Attila Csaba Marosi, Zoltán Balaton, Péter Kacsuk

MTA SZTAKI Computer and Automation Research Institute of
Hungarian Academy of Sciences H-1528 Budapest, P.O.Box 63, Hungary
{atisu,balaton,kacsuk}@sztaki.hu

Abstract

Desktop Grids represent an alternative trend in Grid computing using the same software infrastructure as Volunteer Computing projects, such as BOINC. Applications to be deployed on a BOINC infrastructure need special preparations. However, there are many legacy applications, that have either no source code available or would require too much effort to port. For these applications BOINC provides a wrapper. This wrapper can handle the simple cases and it is configurable, but it can only be used to execute a list of legacy executables (tasks) one after the other. GenWrapper aims to provide a generic solution for wrapping and executing an arbitrary set of legacy applications by utilizing a POSIX like shell scripting environment to describe how the application is to be run and how the work unit should be processed. This is realized by an extended version of BusyBox providing the most common UNIX commands and a POSIX shell interpreter in a single executable with a special applet (BusyBox extension) to make BOINC API functions accessible from the shell on Windows, Linux and Mac OS X platforms. In this paper we present how GenWrapper works and how it can be used to port legacy applications to Desktop Grid systems.

1. Introduction

Desktop Grids (DGs, a branch of Grid computing) are strongly related to Volunteer Computing and are utilizing the same software infrastructure, unlike conventional Service Grids, which are typically built on a complex middleware. As a consequence, installation and maintenance of DG resources (i.e. machines donating resources to the grid) is extremely simple, requiring no special expertise thus, DGs are able to utilize non-dedicated machines. This means that, large number of donors can easily contribute to the pool of shared resources as in Volunteer Computing projects. Unlike Service Grids though, Volunteer Computing projects usually serve only a very limited user community (or target applications) who are able to use the resources for computation. The common architecture of Desktop Grids and Volunteer Computing systems typically consists of one or more central servers and (typically large number of) clients that connect to them time to time. The central servers provide the applications and their input data in form of work units (WUs, i.e. conveniently sized chunks of computation to keep clients busy between contacting the server). Clients can join voluntarily, offering to download and run WUs of an application with a set of input data. When the computation has finished, the client uploads the results to the server where the master part of the application assembles the final output from the partial results returned by clients.

Desktop Grids may be deployed locally within an institution gathering local computing resources (Local Desktop Grid or LDG which is a different goal than Volunteer Computing projects have) or like Volunteer Computing systems gathering resources from the general public (Public Desktop Grid, PDG). One of the most popular software infrastructures is BOINC[1] which aims to provide an open infrastructure for Volunteer Computing Projects and Public Desktop Grids. SZTAKI Desktop Grid[7], which is based on BOINC aims to provide enhancements to fulfill the needs of Desktop Grids including LDGs and to allow more users to access the DG resources.

Any application to be deployed on a BOINC infrastructure needs special preparations. However, there are many so called legacy applications, that have either no source code available to modify or simply would require to much effort to port. For these applications BOINC provides a wrapper which can be used to handle the communication with the Core Client, while executing the legacy application as a subprocess. This wrapper can handle the simple cases but

it is not very flexible. It is configurable, but it can only be used to execute a list of legacy executables (tasks) one after the other. This is because the XML file it uses only allows describing the order of execution of the binaries. To make the wrapper more flexible this configuration file could be extended with new features to provide a required level of flexibility each time a shortcoming is discovered, but ultimately a general solution would require a generic scripting language for describing all possible configuration options.

Realizing this, our GenWrapper aims to provide a generic solution for wrapping and executing an arbitrary set of legacy applications by utilizing a POSIX like shell scripting environment to describe how the application is to be run and how the work unit should be processed. This choice provides great flexibility and a powerful tool to port more legacy applications to Desktop Grids with very little effort. The GenWrapper consists of an extended version of BusyBox[8], a single binary providing essential UNIX commands(such as *sed, grep, unzip, tar, awk*, etc.) and a POSIX shell interpreter (based on *ash*). GenWrapper is ported to run on Windows, Linux and Mac OS X platforms and it is extended to make BOINC API functions accessible from the shell (e.g. *boinc resolve_filename* or *boinc fraction_done*). In the following sections we present how GenWrapper works and how it can be used to port legacy applications to Desktop Grid systems.

2. Applications under BOINC

Volunteers are joining the DG by installing and running the BOINC Core Client and attaching to one or more projects. Besides handling communication with the project servers, the Core Client is responsible for: i) starting, stopping, suspending and resuming the application; ii) enforcing resource limits and resource shares between different projects set by the user; iii) instructing the application to checkpoint itself and iv) accepting various statistics reported by the application (its completion percentage, used CPU time).

To use the distributed resources gathered by BOINC, the application performing the computation also needs some preparation to be able run on the client machines under the control of the Core Client. Apart from having executables for all possible platforms that are member of the DG, the application also has to be prepared to be run by the BOINC Core Client which has two main aspects: i) it should be able to run in the directory structure used by the client, i.e. application executables are placed in the project directory while the working directory is a separate slot directory where input and output files are linked; ii) it should be able to interact with the Core Client, i.e. handle suspend, resume and quit requests and report used CPU time and checkpoints.

BOINC provides an API that applications should use to communicate with the Core Client and handle running in the DG environment. This API provides functions for resolving links to files that are accessed from the slot directory, communicate exit status to the Core Client so it can handle errors and report statistics as needed.

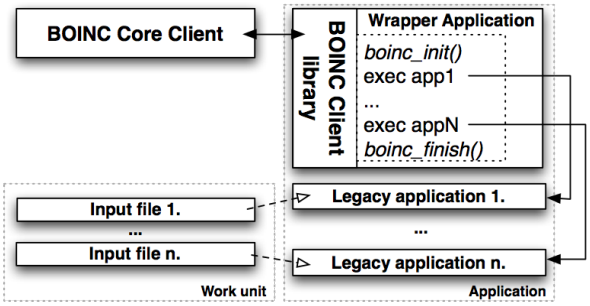

Figure 1. Legacy application using the BOINC Wrapper

Legacy applications or applications which cannot be modified to use the API are not able to run under BOINC because without calling the right API functions they would find links instead of their input files, write their outputs to the wrong place and without properly reporting statistics to the Core Client the application would be restarted over and over and eventually it would be marked as failed. For these applications BOINC offers the BOINC Wrapper (see Figure 1) which acts as a main program managing communication with the Core Client calling the appropriate API functions and running the real application executable as a subprocess. An application using the BOINC Wrapper contains the wrapper executable besides the application files.

3. Applications with GenWrapper

A typical GenWrapper application consists of a zip file holding all the files belonging to legacy application(s), the two GenWrapper components: GitBox and Launcher executables and a *profile script* to perform platform specific preparations. A typical work unit for a GenWrapper wrapped application contains the input files and another shell script (the *work unit shell script*), which allows to control and execute the legacy applications in an arbitrary manner for each work unit. The work unit shell script should be platform independent as the work unit can be executed by any supported platform.

GitBox is a stripped-down Windows only port of BusyBox originally created for the Windows version of the git [9] version control system which internally relies on running shell scripts. Although GIT on Windows later abandoned GitBox, this port was used as the basis for GenWrapper af-

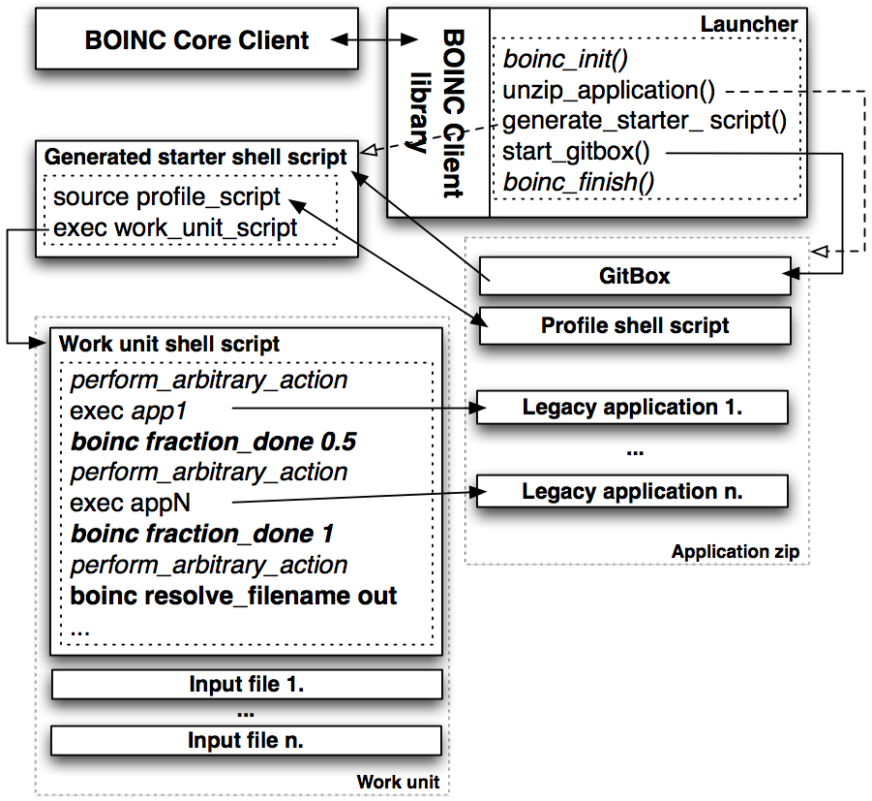

Figure 2. Legacy application using GenWrapper

ter extracting it from GIT and porting back to UNIX preserving functionality on Windows. Later the GenWrapper GitBox version was updated to match newer BusyBox releases, and thus diverged from the original GitBox significantly, although it is still referred to as GitBox in the GenWrapper distribution for historical reasons. Besides Windows the GenWrapper GitBox (which we will simply call GitBox now) is also supported on Linux and Mac OS X, some of the stripped down parts were put back and new BusyBox functionality (e.g. lzma compression) were added and it was extended with BOINC specific shell commands.

A GenWrapper wrapper legacy application is executed as follows. The client downloads the Launcher executable (named like the application as BOINC expects), an application zip file and an optional profile script as the BOINC application and a work unit (input files and a work unit shell script). The Launcher is started by BOINC and acts as a BOINC application, handling all communication with the Core Client. After starting, the Launcher looks for a .zip file with the same name as itself and extracts all files from it to the slot directory. Storing application files in a .zip file is optional, if not found no extracting is performed and the work unit script should access it by resolving its logical name as any other input files. It is recommended to use application zip files, because the BOINC server stores all application files in one common location on the project web server and when files with the same name, but different content are required by different applications a conflict may happen which is prevented by storing application specific files in a zip archive. The most obvious scenario would be that different applications require different versions of the same DLL (Windows shared library) files. These files commonly have the same 8 character filename (plus the ".dll" extension) regardless of their version. Without packaging application files together in a zip, a later deployed application could overwrite the same named files belonging to a previously installed application. The application zip file may also contain a "profile script" which serves as a platform specific bootstrap script for the application (e.g. on Linux the library include path may need to be adjusted or local optimization options could be enabled depending on the presence of optional features, etc.). After unzipping the application archive, the Launcher generates a starter script which first sources the profile script if exists and then executes the work unit shell script. Then Launcher calls the built in POSIX shell interpreter (*ash*) of GitBox which starts to execute this generated script.

The Launcher remains running while GitBox executes

the script and handles communication with the Core Client and performs similar tasks as the BOINC Wrapper. In fact Launcher was originallty based on BOINC Wrapper, but it is heavily modified to fit the needs of GenWrapper. Modifications include: i) suspending and resuming GitBox and all the subprocesses started by it when the Core Client asks for this; ii) measuring and reporting the CPU time used by the running subprocesses and iii) killing the subprocesses if the requested or the client is stopped. The Launcher is spawning a new process for GitBox which is also spawning a new process for each legacy application it is executing. If the functionality of the original BOINC Wrapper were used here, only the GitBox process could be controlled and measured, while loosing control over the legacy application processes (which do the actual work) and the Core Client having no information about them. These tasks are implemented differently on UNIX and Windows systems due to the lack of common API concepts and Windows' limited support of POSIX.

On Linux and Mac OS X there are process groups that the Launcher utilizes by simply putting the spawned GitBox process in a new process group and by default all its child processes will also belong to the same group. The limitation here is that no child process should break away from the process group, thus currently no subshells are supported (scripts should not create background processes or in practice & and parenthesizes should be avoided) and also the legacy applications should avoid create new process groups (although they can spawn subprocesses which are not breaking away from the process group).

On Windows, there are no process groups (a feature with the same name exists, but it is only vaguely similar to UNIX process groups and cannot be used the same way). The closest feature the WIN32 API provides is called JobObject. Each JobObject represents a collection of processes. But the problem with them is twofold: i) by default a process started by a process in a JobObject (child) should also belong to the same JobObject as its parent, but unfortunately it depends which system function was used to create the child process (CreateProcess() is fine, but _spawn() is not always), thus not every child process may end up in the JobObject; ii) there is no official, documented API function to suspend or resume a process, only threads can be controlled; if a process has more than one thread suspending them in the wrong order might lead to a dead-lock. This last problem is solved by using undocumented Windows NTAPI calls that can directly suspend and resume processes. The first problem was overcome by periodically checking the list of running processes whether there is a new one whose parent process belongs to the JobObject but it isn't. If such processes are found they are added to the JobObject. There is no function to suspend or resume all processes in a JobObject, but it is possible to terminate all processes of it. Thus, if suspend or resume is requested the JobObject is queried for the list of its processes and each one of those is handled one by one.

```
IN=‘boinc resolve_filename in‘
OUT=‘boinc resolve_filename out‘
NUM=‘cat ${IN}‘
PERCENT_PER_ITER=$((10000 / NUM))
for i in ‘seq $NUM‘; do
    PC=$((PERCENT_PER_ITER * i /1000))
    boinc fraction_done_percent ${PC}
    echo −e "I␣am␣${PC}%␣complete.␣" >> ${OUT}
    sleep 1;
done
```

Figure 3. Sample GenWrapper work unit shell script

GitBox (as well as BusyBox) has a modular structure, which allows to easily extend it with arbitrary commands by so called "applets". The BOINC extension is implemented in such an applet and currently consists of the most important BOINC API calls such as, *resolve_filename*, *fraction_done*, *fraction_done_percent*. A minimalist sample work unit script for demonstrating the basic capabilities can be seen in Figure 3. This sample is reading an (integer) value from the file with the logical filename 'in' (by first resolving the link to the real file), performing a loop which: i) calculates how much of the total work is done; ii) print the fraction done in percent into the file with logical name 'out' and iii) sleep for a second in each iteration. There is no need to provide or call boinc_init() or boinc_finish() from the script itself, because it is called by the Launcher, which also takes care of forwarding the exit status to boinc_finish (thus the script can normally exit with a non-zero status to signal an error). The Launcher also measures used CPU time and reports to the Core Client automatically. The only BOINC API functionality required for this simple example is to resolve logical filenames and report the fraction done which is also the case for the majority of legacy applications.

4. GenWrapper in Action

GenWrapper has been used to adapt several legacy applications to run in a DG environment within the EDGeS project [2]. The primary goal of the EDGeS project is to build technological bridges to facilitate service and desktop grid interoperability, but the deployment of the developed technologies in a production environment and porting applications to run on the integrated SG—DG environment is also an important part of the project. Hence, the deployed applications are required to run on EGEE [4], BOINC and XtremWeb [5] and need to be adapted for all three plat-

forms. This can be achieved by modifying applications or developing application specific wrappers. Application specific wrappers are turned out to be often necessary because legacy codes are written in a diverse set of languages (e.g. one of them uses the R statistical language), often require application specific setup and/or processing of input and output files (e.g. archive creation, compression or platform specific setup at runtime), and so on. By using GenWrapper the required time and effort to deploy these applications became significantly less by allowing to adapt the applications to BOINC without changing their code and avoiding reimplementing the same functionality over and over in application specific wrappers.

The aims of the CancerGrid project[6] are to develop focused molecule libraries with a high content of anticancer leads and to build models for predicting various properties for the molecules. Processing of the molecules is automated by different computer algorithms (also called *in silico* drug design). These algorithms are implemented by existing legacy applications that are to be run on a DG system. They must be executed in a strict order to produce the desired output therefore workflows have been designed for each procedure such as, molecule descriptor calculation, model building and property prediction. The most computationally intensive workflow is the descriptor calculation. This workflow contains 4 jobs which perform molecular calculations, 2 jobs converting file formats and 3 database manipulator jobs. The workflow has two parameters: N represents the number of two dimensional input molecules, M represents the number of conformers (variants of a molecule) that are generated in a session. The molecular calculation jobs are thus executed by a large number of times: once for each input or once for each conformer of each input molecule (N or $N \times M$). The typical value for N is $30,000$ and for M is 100 thus, the workflow generates $3,000,000$ instances for each job and altogether almost 10 million jobs are generated during the execution of the workflow. The granularity of the workflow is however very fine grained, the typical running time of one instance is a few minutes, so they are not suitable for conversion to BOINC work units one to one.

To optimize the number of work units generated from a workflow, we introduced a job database, job queues and a Queue Manager extension at the BOINC server. The job database stores the incoming jobs generated from different workflow instances into separate queues belonging to the different algorithms. Once a queue contains appropriate number of jobs, the Queue Manager generates a single BOINC work unit out of a batch of jobs and places it into the BOINC database using DC-API[3]. It also generates a shell script to manage the execution of the batch on the client processing the work unit which is executed by GenWrapper. The batch script is generated by concatenating predefined head, body and tail script fragments which handle preparing inputs (i.e. getting the appropriate part from the batch), running the computation, and putting the outputs back in the batch output respectively. These scripts can contain macros denoted by `%{name}` which are substituted by the Queue Manager when generating the batch scripts. Macros are used to denote the intended location and name of input and output files which should be followed for batch execution to work. Besides supporting batch execution GenWrapper is also used to execute an application specific script (embedded in the batch body script fragment) which handles file manipulation/renaming needed to run the legacy application. This is required because some legacy codes require the files to be named according to an identifier which is only determined during execution of the work unit and not known at the time of its creation. Also some of the calculations generate an unknown number of output files which is handled by putting them in a compressed archive

```
BASEDIR=‘pwd‘
gzip -cd ‘boinc resolve_filename %{inputs}‘ \
        | tar xvf -
exec 3<&2
exec 2>"$BASEDIR"/%{output_dir}/gridnfo.log
touch "$BASEDIR"/%{output_dir}/gridnfo.log
WORKFILE_PREFIX="WORKFILE"
MODEL_TYPE="Multilinear␣Regression"
cd "$BASEDIR"/%{input_dir}
mv IN-PARAMS IN-PARAMS.ori
echo "EXE_DIRECTORY␣=␣$BASEDIR" >IN-PARAMS
echo "WORKFILE_PREFIX␣=␣$WORKFILE_PREFIX" \
        >>IN-PARAMS
echo "MODEL_TYPE␣=␣$MODEL_TYPE" >>IN-PARAMS
cat IN-PARAMS.ori | grep -v "EXE_DIRECTORY" \
        | grep -v "WORKFILE_PREFIX" \
        | grep -v "MODEL_TYPE" >>IN-PARAMS
rm IN-PARAMS.ori
mv IN-MATRIX $WORKFILE_PREFIX.mt
export LD_LIBRARY_PATH="$BASEDIR"
"$BASEDIR"/mda -p IN-PARAMS -r
mv $WORKFILE_PREFIX.mt IN-MATRIX
mv $WORKFILE_PREFIX.stdout \
        "$BASEDIR/%{output_dir}/stdout.log"
mv $WORKFILE_PREFIX.stderr \
        "$BASEDIR/%{output_dir}/stderr.log"
mv $WORKFILE_PREFIX.model \
        "$BASEDIR/%{output_dir}/OUT-MODEL"
exec 2<&3
exec 3>&-
cd "$BASEDIR"
for i in ‘find %{output_pattern} \
        -name gridnfo.log‘; do
        cat $i
done 1>&2
tar cf - %{output_pattern} | \
        gzip >‘boinc resolve_filename %{outputs}‘
```

Figure 4. A typical CancerGrid work unit shell script

created by GenWrapper after the legacy application is finished.

The flexibility provided by GenWrapper was used fully to adapt the legacy applications without modifying their code and to handle the above mentioned issues with only writing relatively simple shell scripts (see Figure 4), which made it easy to support the advanced scenario described above.

5. Conclusion

GenWrapper offers a generic solution for wrapping and executing an arbitrary set of legacy applications in a BOINC infrastructure. The main strength of it is the POSIX like shell scripting environment which is used to describe how the application is to be run and how the work unit should be processed. This provides great flexibility and a powerful tool to port legacy applications to Desktop Grids with very little effort.

GenWrapper is based on GitBox which is a stripped down version of BusyBox that was ported to run on Windows. We ported GitBox back to UNIX (Linux and Mac OS X) and extended it with BOINC specific commands and a Launcher component that handles starting GitBox and communicating with the Core Client such as reporting CPU time and handling Suspend/Resume requests, much like the BOINC Wrapper does. Currently the GitBox and Launcher components of GenWrapper are two separate executables, each about 400 kiB size but we plan to integrate the two in a single executable which will probably make it even smaller.

GenWrapper has been used to adapt several legacy applications to DGs. It was used by the CancerGrid project which is utilizing DGs for drug development. In this project computationally intensive workflows consisting of many jobs were created which required running legacy applications on a DG system. Porting these applications would have require much effort, and the functionality of the original BOINC wrapper could not be used to do it (some legacy codes produce variable number of output files that cannot be predicted before running them, some need special preparations before execution, most of the files are text files that need to be decompressed/compressed before and after processing, etc.). GenWrapper is also used by the EDGeS[2] project which is building technological bridges to facilitate service and desktop grid interoperability and porting applications to run on this integrated environment. The deployed applications are required to run on EGEE, BOINC and XtremWeb. By using GenWrapper the required time and effort to deploy these applications became significantly lower and the flexibility allowed by POSIX scripting helped to support complex cases as in CancerGrid easily.

All parts of GenWrapper are open source and are available for download upon request.

6. Acknowledgements

The results presented in this paper are realized with the support of the Enabling Desktop Grids for e-Science (EDGeS) project, (co-founded by the European Commission under contract number RI-211727) and by the CancerGrid project (co-founded by the European Commission under contract number 037559).

References

[1] D. P. Anderson. BOINC: A system for public-resource computing and storage. In *Proc. of 5th IEEE/ACM International Workshop on Grid Computing*, Pittsburgh, USA, November 2004.

[2] Z. Balaton, Z. Farkas, G. Gombás, P. Kacsuk, R. Lovas, A. C. Marosi, A. Emmen, G. Terstyánszky, T. Kiss, I. Kelley, I. Taylor, O. Lodygensky, M. Cardenas-Montes, G. Fedak, and F. Araujo. EDGeS: The common boundary between service and desktop grids. *Parallel Processing Letters*, 18(3):433–445, September 2008.

[3] Z. Balaton, G. Gombás, P. Kacsuk, A. Kornafeld, J. Kovács, A. C. Marosi, G. Vida, N. Podhorszki, and T. Kiss. SZTAKI Desktop Grid: a modular and scalable way of building large computing grids. In *Proc. of the 1st Workshop on Desktop Grids and Volunteer Computing Systems in conjunction with IPDPS'07*, Long Beach, CA, USA, March 2007. IEEE.

[4] EGEE Enabling Grids for E-SciencE. `http://www.eu-egee.org`.

[5] G. Fedak, C. Germain, V. Néri, and F. Cappello. Xtremweb: A generic global computing system. In *Proc. of CCGRID2001 Workshop on Global Computing on Personal Devices*. IEEE Press, May 2001.

[6] P. Kacsuk, K. Karóczkai, G. Hermann, G. Sipos, and J. Kovács. WS-PGRADE: Supporting parameter sweep applications in workflows. In *Proc. of 3rd Workshop on Workflows in Support of Large-Scale Science in conjunction with SC'08*, Austin, TX, USA, November 2008.

[7] A. C. Marosi, G. Gombás, Z. Balaton, P. Kacsuk, and T. Kiss. SZTAKI Desktop Grid: Building a scalable, secure platform for desktop grid computing. In *Making Grids Work*, pages 363–374. Springer Publishing Company, Incorporated, July 2008.

[8] B. Perens and al. Busybox: The swiss army knife of embedded linux. `http://www.busybox.net`.

[9] L. Torvalds and al. Git: Fast version control system. `http://git.or.cz/`.