Principles of Transactional Grid Deployment

Brian Coghlan, John Walsh, Geoff Quigley, David O'Callaghan, Stephen Childs, and Eamonn Kenny

Department of Computer Science, Trinity College, University of Dublin.
`ekenny@cs.tcd.ie, coghlan@cs.tcd.ie, john.walsh@cs.tcd.ie, stephen.childs@cs.tcd.ie, david.ocallaghan@cs.tcd.ie, geoff.quigley@cs.tcd.ie`

Abstract. This paper examines the availability of grid infrastructures, describes the principles of transactional grid deployment, outlines the design and implementation of a transactional deployment system for the Grid-Ireland national grid infrastructure based on these principles, and estimates the resulting availability.

1 Introduction

Consider for instance that a new release of some infrastructural grid software is incompatible with the previous release, as is often the case. Once a certain proportion of the sites in a grid infrastructure are no longer consistent with the new release then the infrastructure as a whole can be considered inconsistent. Each grid infrastructure will have its own measures, but in all cases there is a threshold below which proper operation is no longer considered to exist. The infrastructure is no longer available. Thus availability is directly related to consistency. An inconsistent infrastructure is unavailable. As yet the authors are not aware of any prior availability estimates for existing grid infrastructures such as LCG[1], EGEE[2] or CrossGrid[3]. This is understandable in these early days of full deployment. Below we examine this subject, present some early estimates, propose a methodology, transactional deployment, that is intended to maximize the infrastructure availability, describe an implementation, and estimate the effect of the methodology on availability.

1.1 Consistency of a Single Site

Assume N identical sites are being evaluated in independent experiments, and that at time t, $C(t)$ are consistent (have upgraded) and $I(t)$ are inconsistent (have yet to upgrade). The probabilities of consistency and inconsistency are:

$$P_C(t) = C(t)/N \qquad P_I(t) = I(t)/N \tag{1}$$

where since $C(t) + I(t) = N$, then $P_C(t) + P_I(t) = 1.0$

The per-site upgrade rate is the number of upgrades per unit time compared with the number of sites that remain inconsistent:

$$U(t) = I(t)^{-1} dC(t)/dt \tag{2}$$

This is the instantaneous upgrade rate $I(t)^{-1}\Delta C(t)/\Delta t$ for one site, i.e. the proportion that upgrade per unit time. Hence the average time a site waits before it becomes consistent (the mean time to consistency MTTC) is:

$$\mathrm{MTTC} = \int_0^\infty P_I(t)dt \tag{3}$$

In some well regulated cases, e.g. 24×7 Operations Centres, it might be that the upgrade rate is close to a constant, say λ. In these cases the probability of a site being inconsistent (not upgraded) at time t and the resulting MTTC are:

$$P_I(t) = e^{-\lambda t} \qquad \mathrm{MTTC} = \int_0^\infty P_I(t)dt = \lambda^{-1} \tag{4}$$

Bear in mind that all the above is true only if the upgrade rate is a constant. However, few organizations can afford facilities like 24×7 Operations Centres, so the general case is that the per-site upgrade rate is not constant, but a fitful function that reflects working hours, holidays, sick leave, etc. Moreover, upgrades at different sites are not independent, since it is usual that sites benefit from the experiences of sites that upgrade before them, but let us ignore this complication.

1.2 Consistency of a Multi-Site Infrastructure

If the infrastructure contains M types of sites, then the Law of Probability states that the probability of all sites being inconsistent (none upgrading) or consistent (all upgraded) for time t is:

$$P_{I\mathrm{infra}}(t) = \prod_{m=1}^{M} P_{Im}(t) \qquad P_{C\mathrm{infra}}(t) = \prod_{m=1}^{M} P_{Cm}(t) \tag{5}$$

First let us take the case where all sites must be consistent for the infrastructure to be considered consistent. Then the probability of an inconsistent multi-site infrastructure is the probability of at least one site being inconsistent, i.e. the probability of NOT(all upgraded):

$$P_{I\mathrm{infra}}(t) = 1 - P_{C\mathrm{infra}}(t) = 1 - \prod_{m=1}^{M}(1 - P_{Im}(t)) \tag{6}$$

The more sites, the greater the probability. Clearly it is easier to understand the situation if all M sites are identical, i.e. the infrastructure is homogeneous, as is the case for Grid-Ireland. The MTTC is:

$$\mathrm{MTTC}_{\mathrm{infra}} = \int_0^\infty P_{I\mathrm{infra}}(t)dt \neq \lambda_{\mathrm{infra}}^{-1} \tag{7}$$

Note λ_{infra} is certainly not a constant, and the more sites, the longer the MTTC.

Now let us take the case where not all sites must be consistent for the infrastructure to be considered consistent. This might be the case for those grids where there are a set of core sites, or those where some inconsistency is acceptable. Here we can use the Binomial expansion:

$$\prod_{m=1}^{M} (P_{Im}(t) + P_{Cm}(t)) = 1.0 \tag{8}$$

where $P_{Cm}(t) = 1 - P_{Im}(t)$. For M identical sites, then $(P_I(t) + P_C(t))^M = 1.0$ where $P_C(t) = 1 - P_I(t)$. For example, for $M = 4$:

$$P_I^4(t) + 4P_I^3(t)P_C(t) + 6P_I^2(t)P_C^2(t) + 4P_I(t)P_C^3(t) + P_C^4(t) = 1.0 \tag{9}$$

The first term represents the probability that no site has upgraded by time t, the second that any 1 has upgraded, etc. The probability of an inconsistent infrastructure representing ≥3 inconsistent sites, and the resulting MTTC, are:

$$\begin{aligned} P_{I\text{infra}}(t) &= P_I^4(t) + 4P_I^3(t)P_C(t) = 4P_I^3(t) + 3P_I^4(t) \\ \text{MTTC}_{\text{infra}} &= \int_0^\infty P_{I\text{infra}}(t)dt \neq \lambda_{\text{infra}}^{-1} \end{aligned} \tag{10}$$

i.e. again, λ_{infra} is not a constant.

Thus if one knows the deployment behaviour of each site it is possible to calculate the MTTC of the infrastructure, especially if one uses symbolic algebra tools such as Maple or Mathematica.

1.3 Mean Time Between Releases

The interval between releases MTBR is quite independent of the MTTC. The MTTC is a deployment delay determined by the behaviour of the deployers, whilst the MTBR is dependent upon the behaviour of developers. Otherwise similar considerations apply.

1.4 Availability of a Multi-Site Infrastructure

The availability of the infrastructure is given by the following equation:

$$A_{infra} = \text{MTBR}_{infra}/(\text{MTBR}_{infra} + \text{MTTC}_{infra}) \tag{11}$$

1.5 Estimates from Testbed Experience

Figure 1 shows derived histograms of the release intervals and deployment delays for the CrossGrid production and development testbeds. The use of this data is open to debate. Firstly one could argue for compensation for 8 hours of sleep per day. Secondly they are derived from email arrival times, not from actual event times. Steps have now been taken by the CrossGrid deployment team in FZK to log these events, so in future much more accurate data will be accessible.

The release intervals greater than 30,000 seconds (approximately 21 days) can be considered as outliers. Two intervals relate to the month of August, a traditional holiday month in Europe. Similarly the deployment delays greater than 3,000 seconds (approximately 2 days) might be considered outliers.

If we ignore the outliers, then from these figures we may derive a notional observed MTBR, MTTC and availability for the CrossGrid production and development testbeds, assuming consistency after the average deployment delay. Thus the observed infrastructure availabilities are as shown in Tables 1 and 2.

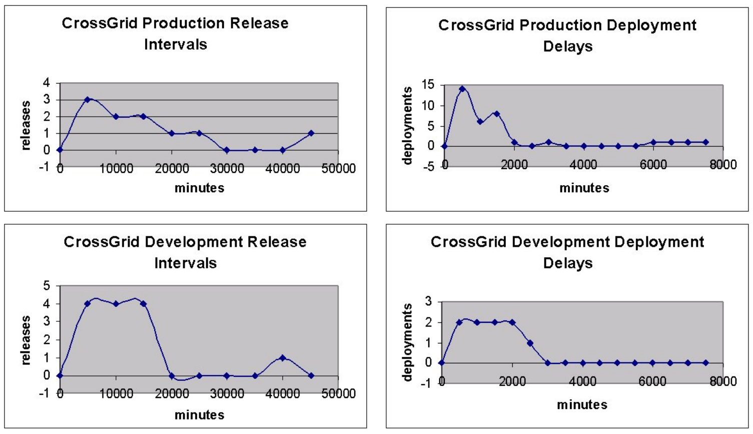

Fig. 1. CrossGrid testbed release intervals and deployment delays

Observed MTBR	163 hours
Observed MTTC	11.5 hours
Observed availability	93.4%

Table 1. Observations for the CrossGrid production testbed

Observed MTBR	120 hours
Observed MTTC	18 hours
Observed availability	86.9%

Table 2. Observations for the CrossGrid development testbed

2 Principles

2.1 Two-Phase Transactionality

Maximizing availability means maximizing the proportion of time that the infrastructure is entirely consistent. This requires either the the time between releases MTBR to be maximized or the MTTC to be minimized. The MTBR is beyond the control of those who manage the infrastructure. On the other hand, if the

MTTC can be minimized to a single, short action across the entire infrastructure then the availability will indeed be maximized.

However, an upgrade to a new release may or may not be a short operation. To enable the upgrade to become a short event the upgrade process must be split into a variable-duration prepare phase and a short-duration upgrade phase, that is, a two-phase commit. Later in this paper we will describe how this can be achieved.

If the entire infrastructure is to be consistent after the upgrade, then the two-phase commit must succeed at all sites. Even if it fails at just one site, the upgrade must be aborted. Of course this may be done in a variety of ways, but from the infrastructure managers' viewpoint the most ideal scenario would be that if the upgrade is aborted the infrastructure should be in the same state as it was before the upgrade was attempted, that is, the upgrade process should appear to be an atomic action that either succeeds or fails.

Very few upgrades will comprise single actions. Most will be composed from multiple subactions. For such an upgrade to appear as an atomic action requires that it exhibits transactional behaviour, that all subactions succeed or all fail, so that the infrastructure is never left in an undefined state.

Thus we can see that to maximize availability requires that an upgrade be implemented as a two-phase transaction.

2.2 Preservation of ACID Properties

Since Gray[4,5], transactions have been widely assumed to preserve so-called ACID properties. Let us consider each of these:

(a) *Atomicity:* if a transaction contains multiple sub-actions, these are treated as a single unit. Either all sites upgrade or none.
(b) *Consistency:* a transaction never leaves the infrastructure in an undefined state. It is enforced by a single distributed commit action in the case of success, or a distributed rollback in the case of an error.
(c) *Isolation:* transactions are separated from each other until they're finished. We only permit one transaction to take place at a time, bypassing this issue.
(d) *Durability:* once a transaction has been committed, that transaction will persist even in the case of an abnormal termination.

2.3 Homogeneous Core Infrastructure

A homogeneous core infrastructure is not strictly required for transactional deployment but is highly desirable. It greatly simplifies the understanding of the infrastructure availability, as can be seen from equations [7–10]. Logically, it allows a uniform control of the grid infrastructure that guarantees uniform responses to management actions. It also assures a degree of deterministic grid management. Furthermore it substantially reduces the complexity of the release packaging and process. Minimizing this complexity implies maximizing the uniformity of the deployed release, i.e. the core infrastructure should be homogeneous.

2.4 Centralized Control via Remote Management

Centralized control of the core grid infrastructure enables simpler operations management of the infrastructure. Remote systems management enables low-level sub-actions to be remotely invoked if a transaction requires it. It also enables remote recovery actions, e.g. reboot, to be applied in the case of deadlocks, livelocks, or hung hardware or software. Realistically, all infrastructure hardware should be remotely manageable to the BIOS level.

3 Transaction algorithm

The transaction algorithm is:

```
ssh-add keys                                      //server startup
fork ssh tunnel(localhost:9000,repository:9000)
file read sitelist
XMLRPC server start
create lock                                       //transaction start
if(lock success) {
  set global_status (preparing)                   // Server prepare
  set release_dir(date, tag)
  cvs_checkout(date, tag)
  if (success) {                                  // prepare sites
    foreach(site in selected_sites) {
      sync RPMS, LCG-CVS, GI-CVS
      backup current GI,LCG links in release_dir
      backup profiles link
      create new GI,LCG links
      build site profile
    }
    if(failure) {                                 //rollback
      foreach(site in selected_sites) {
        restore link backup from release_dir
      }
      if( rollback_failure )
        set status rollback_error
    } else {                                      // commit
      foreach(site in selected_sites) {
        create new profiles link
} } } }
delete lock                                       //transaction end
```

4 Implementation

4.1 Architecture

Logically, the implementation divides into three segments. The first is the repository server, which hosts both the software to be deployed and the Transactional

Deployment Service (TDS) logic. Secondly, there is the user interface, which has been implemented as a PHP page residing on an Apache web server. Finally there are the install servers at the sites that we are deploying to. These servers hold configuration data and a local copy of software (RPMs) that are used by LCFGng to maintain the configuration of the client nodes at the site. It is the state of these managed nodes that we are trying to keep consistent.

4.2 Communication

The TDS establishes connections to the web server and the various install servers using SSH. The TDS is run within ssh-agent and it loads the keys it needs at start-up. Firewall rules on the repository server, where the TDS resides, and the web server, where the user interface resides, prevent unwanted connections. Susceptibility to address spoofing attacks is minimized by only allowing the TDS to create network connections.

The use of ssh-agent is very powerful, as it allows the TDS software to securely initiate multiple outbound connections as and when they are needed without any need for the operator to enter passwords. This ability was a key enabler in allowing all the control to come from the central server without having to push scripts out to each site. It also means that the TDS directly receives the return codes from the different commands on the remote sites. For extra security, when the remote sites are configured to have the TDS in their authorized keys files, the connections are restricted to only be accepted from the repository server, to disallow interactive login and to disallow port tunnelling (an important difference between the configuration of the ssh connection to the web server and to the remote sites!).

4.3 Transactional Deployment Server

The TDS consists of a Perl script and a small number of shell scripts. Thanks to the use of high level scripting languages the total amount of code is relatively small — approximately 1000 lines. Much of this code is in the main Perl script, which runs as an XMLRPC server[6,7]. There are two reasons for the choice of XMLRPC over HTTP for the communications between the TDS and the user interface. Firstly, the XMLRPC protocol is not tied to a particular language, platform or implementation. There are numerous high quality and freely available implementations and they mostly work together without problems. Secondly, XMLRPC is simple and lightweight. Various other solutions were considered, such as SOAP communications with WS-Security for authenticating the communications at the TDS. However, WS-Security is still in development, and SOAP interoperability is in debate within the WS-I initiative.

XMLRPC exposes methods for preparing a site, committing changes and rolling back changes. Internally, the TDS uses a series of shell scripts to perform the actions associated with the various stages. There are a small number of these groups of actions to collect together as a single atomic prepare phase. Importantly, each action that takes place is checked for success and if any action fails then the whole group of actions returns an error. The error code returned can be used to determine exactly where the failure occurred. Only one softlink needs to be changed to roll back a remote site. The other changes that have taken place are not rolled back; this results in mild clutter on the remote file system, but does not cause significant problems. If the changes to the softlinks are not rolled back there is a possibility of incorrect configurations being passed through to the LCFG clients should an administrator attempt to manually build and deploy profiles. We intend to add extra functionality so multiple past transactions can be rolled back.

Locks are used to prevent more than one transaction being attempted at a time. Should the TDS exit mid-transaction, the lock file will be left in place. This file contains the process id of the TDS, so that the system administrator can confirm that the process has unexpectedly exited. There is no possibility of deadlock.

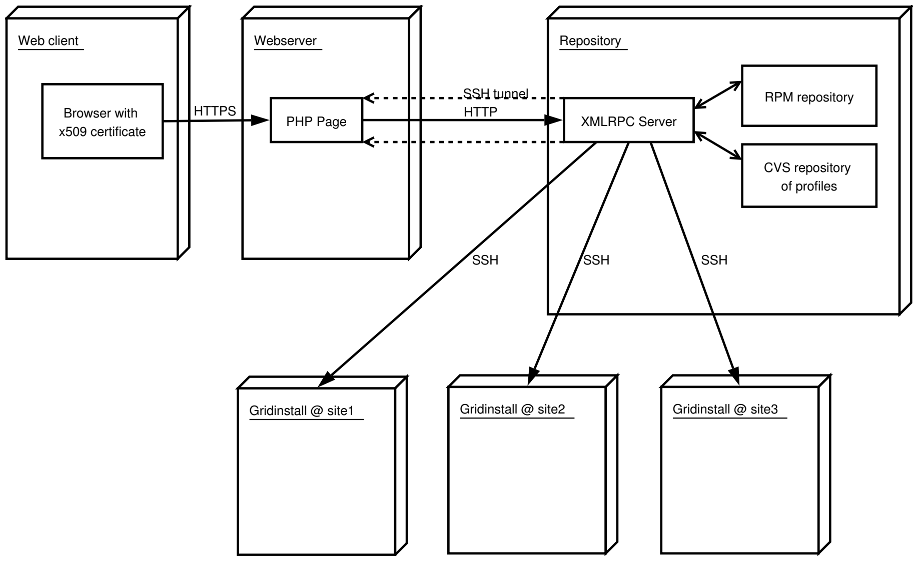

Fig. 2. Transactional Deployment System architecture

4.4 User interface

The user interface is a simple page implemented in PHP and deployed on an Apache web server. Access to the page is only allowed via HTTPS and Apache is configured to only accept connections from browsers with particular certificates installed. The PHP code uses XMLRPC to communicate with the TDS to determine which sites are available for deployment and what their current state is. A drop-down list is populated with a list of the available version tags from the Grid-Ireland CVS repository. These tags are only set for known good releases of configuration settings. As previously stated, the TDS establishes an

SSH tunnel between the repository server and the web server, so the PHP code opens communications to a local port on the web server.

Users of the interface have five actions available. The simplest of these is to simply update the information displayed. The other actions are *prepare*, *commit*, *rollback* and a combined *prepare and commit* action. For the prepare actions, the user needs to select a release tag to deploy plus the sites to deploy to. Once the prepare action has been performed the control for selecting a release is disabled until either a commit or rollback has taken place. Commit and rollback do, however, accept selection of which sites to commit or rollback. Failure in the prepare phase will normally result in an automatic rollback. The normal usage of the system will be to select sites and a release tag, and then to use the combined prepare and commit action. This reduces the time between the prepare and commit phase, reducing the likelihood of problems appearing in this time, and ensuring that prepare and commit are carried out on the same list of sites.

5 Evaluation

It will be a while before statistically significant data is available for transactional deployment to a real grid infrastructure such as Grid-Ireland. The MTBR, i.e. the time between releases, is the same with or without transactional deployment. Let us assume the CrossGrid production testbed MTBR by way of example.

The transactional deployment delay is the sum of three consecutive delays: the commit time T_{commit}, the time T_{signal} for the LCFG install server to successfully signal its client nodes, and the time T_{update} it takes for a client node to update to the new release once it has recognized the server's signal. Our current estimates of these are: $T_{commit} = 20mSec$, $T_{signal} = 10minutes$ worst case, and $T_{update} = 7.5minutes$ worst case. The worst case value for T_{signal} results from the fact that client nodes check for the update signal on a 10 minute cycle. The worst case value for T_{update} is an average of 50 runs of a clean installation of a LCFG Computing Element, so it represents an extreme upper bound on representative physical machines. Therefore the worst-case MTTC is 17.5 minutes. The resulting worst-case infrastructure availability is shown in Table 3.

Estimated MTBR	163 hours
Estimated MTTC	17.5 minutes
Estimated availability	99.82%

Table 3. Example MTBR, MTTC and availability for transactional deployment

If the LCFG client nodes could be signalled to update immediately, i.e. $T_{signal} = 0$, then the availability would be increased 99.92%, and in fact this is likely to be substantially better for realistic release updates.

6 Conclusions

It is clear that the infrastructure has greatly enhanced availability with transactional deployment, improved from 87–93% to 99.8%, with the potential for 99.9% availability with a small amount of extra work.

The concept is certainly scalable to many tens of sites, but probably not hundreds, and almost certainly not at this time across national boundaries. How can this be accommodated? One obvious solution is to mandate compatibility across releases. The advantage is that a national commit is no longer necessary, although a site commit is still worthwhile to reduce the MTTC. The disadvantage is that it is very restrictive for developers. Another possibility is to avail of the evolving federated structures within international grids such as EGEE. Transactional deployment could be performed in concert across a federation by the relevant Regional Operating Centre [8], perhaps loosely synchronized across national boundaries. The mooted International Grid Organization could then act as a further level of loose synchronization between the Core Infrastructure Centres of each federation.

Without doubt the most important benefit of the transactional deployment system not yet mentioned is the ease it brings to deployment. Transactional deployment allows for an automated totally-repeatable push-button upgrade process, with no possibility of operator error. This is a major bonus when employing inexperienced staff to maintain the grid infrastructure.

References

1. LCG: Large hadron collider computing grid project. http://lcg.web.cern.ch/LCG/ (2004)
2. EU: Enabling grids for e-science in europe (EGEE). http://www.eu-egee.org/ (2004)
3. EU: Crossgrid. http://www.crossgrid.org/ (2004)
4. Gray, J.N.: Notes on data base operating systems. In Bayer, R., Graham, R., Seegmuller, G., eds.: Operating Systems: An Advanced Course. Springer Verlag, New York, NY (1978) 393–481
5. Gray, J., Reuter, A.: Transaction Processing: Concepts and Techniques. Morgan Kaufman (1993)
6. Userland-Software: XML-RPC home page. http://www.xmlrpc.org (2004)
7. Ray, R.J., Kulchenko, P.: Programming Web Services with Perl. O'Reilly (2003)
8. EGEE: SA1: Grid operations. http://egee-sa1.web.cern.ch/egee-sa1 (2004)