SECURE APPLICATION DEPLOYMENT IN THE HIERARCHICAL LOCAL DESKTOP GRID

Attila Csaba Marosi, Gábor Gombás and Zoltán Balaton
MTA SZTAKI
Laboratory of Parallel and Distributed Systems
atisu@sztaki.hu, gombasg@sztaki.hu, balaton@sztaki.hu

Abstract   The Desktop Grid model harvests the unused CPU cycles of any computer connected. In this paper we present a concept how the separated Desktop Grids can be used as building blocks for larger scale grids by organizing them in a hierarchical tree. We present a prototype implementation and show the challenges and security considerations we discovered. We describe methods and give solutions how security can be enhanced to satisfy the requirements for real-world deployment.

1.   Introduction

Contrary to traditional grid[11] systems where the maintainers of the grid infrastructure provide resources where users of the infrastructure can run their applications, desktop grids provide the applications and the users of the desktop grid provide the resources. Thus, a major advantage of desktop grids is that they are able to utilize a huge amount of resources that were not available for traditional grid computing previously.

Users of scientific applications usually are concerned only about the amount of computing power they can get and not about the details how a grid system delivers this computing power. Therefore, they want to develop a single application that in turn can run on any infrastructure that provides the most appropriate resources at a given time. Unfortunately existing applications have to be modified in order to run on desktop grid systems and this makes desktop grids less attractive for application developers than traditional grid systems.

2.   Desktop Grids

The common architecture of desktop grids consists of one or more central servers and a large number of clients. The central server provides the applications and their input data. Clients join the desktop grid voluntarily, offering to download and run an application with a set of input data. When the application has finished, the client uploads the results to the server. Based on the environment where the desktop grid is deployed we must distinguish between two different concepts.

Global Desktop Grids

Global Desktop Grids (also known as Public Desktop Grids) consist of a publicly accessible server providing projects and the attached clients. There are several unique aspects of this computing model compared to traditional grid systems. First, clients may come and go at any time, and there is no guarantee that a client that started a computation will indeed finish it. Furthermore, the clients cannot be trusted to be free of either hardware or software defects, meaning the server can never be sure that an uploaded result is in fact correct. Therefore, redundancy is often used by giving the same piece of work to multiple clients and comparing the results to filter out corrupt ones.

Local Desktop Grids

To fill the gap between the traditional grids and the desktop grids SZTAKI introduced the concept of Local Desktop Grids. Local Desktop Grids are intended for institutional or industrial use. Especially for businesses it is often not acceptable to send out application code and data to untrusted third parties (sometimes this is even forbidden by law). The project and clients are shielded from the world by firewalls or any other means. This environment gives more flexibility by allowing the clients to access local resources securely and since the resources are not voluntarily offered the performance is more predictable.

SZTAKI Local Desktop Grid

As we can see there is a huge difference between traditional grids and desktop grids. We also have to make a distinction between the publicly used Global Desktop Grids and the Local Desktop Grid concept. The SZTAKI Local Desktop Grid[4] (or SZTAKI LDG) implements the latter. It is based on BOINC[1] technology and extends it according to the needs of institutional and business users. BOINC is originating from the SETI@Home[3] project to provide an open infrastructure for utilizing the computers of people interested in the outcome of a project.

We faced several possibilities when designing SZTAKI LDG: to develop our own solution[15], to use other desktop grid systems and approaches like Distributed.net[9], Legion[13], JXTA[8] or Entropia[10]. We decided to build on BOINC, because it has a large user base, it's open source, cross-platform and has a clean design[2] and implementation making it the best target for (third party) enhancements[5].

SZTAKI Desktop Grid has a Public Desktop Grid version[6] running currently with more than 12000 registered users.

3. Hierarchy

One of the enhancements of the SZTAKI Local Desktop Grid hierarchy. Hierarchy allows the use of desktop grid projects as building blocks for larger grids, for example divisions of a company or departments of a university can form a company or faculty wide desktop grid. Every project has a classical parent-child relationship with the others. They may request work from a project above (*consumer*) or may provide work for a project below (*producer*). The project server can enter a hierarchical mode, when one of it's consumers require more work than it has for disposal. It will then contact one of it's producer nodes and request more work.
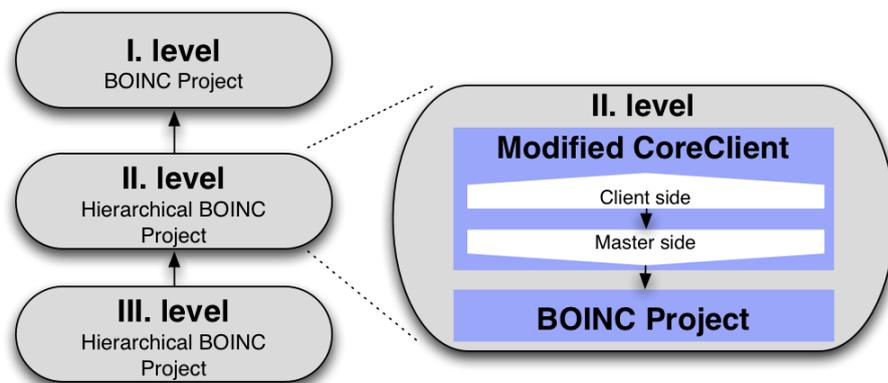

Figure 1. The split architecture of the Hierarchy prototype. Inside the *CoreClient* the *Client side* is acting as a consumer by requesting work and the *Master side* as a producer by providing work for the project.

It is allowed for a project to have more consumers and producers. We use the simple layout in *Figure 1* for presenting the enhancements of the architecture only. The architecture consists of a modified BOINC CoreClient and a project. The CoreClient is required normally for the clients to participate in any project,

but now the CoreClient is running on the machine hosting the consumer project. Originally it's task is also to dynamically download the application of the project which is doing the actual computation for the project.

BOINC terminology uses the *platform* expression for the specific combinations of architectures and operating systems. We modified the CoreClient such that we may specify what platform it should pretend to be using. This allows us to query all the predefined platforms for applications however, the deployment of the application on the lower levels is not handled by the CoreClient, it is the task of the project administrator. When the number of unsent workunits runs below a specified threshold the CoreClient will contact a producer for work. A project may have more producers configured each with a priority assigned. First the producer with the highest priority will be contacted for work, if it fails to provide work then the next one is queried and so on. The modified CoreClient has a split architecture, first it consumes work from a producer (*Client side*), second it injects the requested work in the local consumer project, thus acts as a provider (*Master side*).

To test our prototype we deployed a seven-level hierarchical environment with clients attached to the lowest level. Six of the servers were running on Debian Linux 3.1/Intel, one was using Mac OS/X 10.4/PowerPC. Six clients were attached, three using Debian Linux 3.1/Intel, two Windows XP and one Mac OS/X 10.4/PowerPC. We also created a simple application for all *platform*s, with the only purpose to produce high load and run exactly for the time given in the *workunit*. Our goal with this diverse environment was not to measure performance, simply to test the environment for possible problems and bottlenecks. Using the prototype we were able to provide basic hierarchical functionality without modifying existing projects. Only a modified CoreClient was needed for work request and distribution. However, this method does not solve issues with the automatic deployment of applications coming form a higher level of the hierarchy, the exponentially growing number of workunits caused by redundancy or any security considerations.

4.   Challenges and solutions

We discovered various problems which the prototype described in the previous section could not solve. In order to be able to provide a model which is mature enough for industrial or institutional deployment the following issues need to be addressed.

Redundancy and deadline

Redundancy ensures every workunit will have a correct result by simply sending the same piece of work to multiple clients and comparing the results to filter out corrupt ones. *Figure 2* shows a three level layout with the redundancy
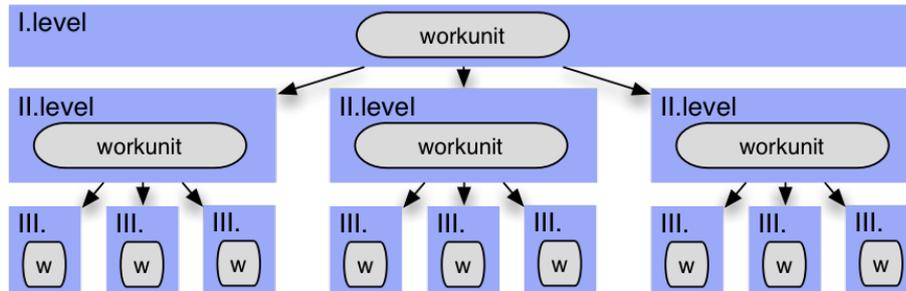
Figure 2.   Growing number of redundant workunits in the hierarchy demonstrated with a simple three level layout.

of three on each level. In this case each producer on each level creates three copies of any workunit received. By the third level there will be nine redundant ones. This means that nine clients will compute the same workunit instead of the supposed three (which was the requested redundancy on the first level). If more levels are added to the hierarchy this number will exponentially grow. It can be solved easily by forcing redundancy to be disabled on all but the first level. This way exactly the requested number of redundant workunits will be distributed.

Deadline is to prohibit workunit-hijacking by clients. Set when the workunit is downloaded, after it passes the workunit is considered invalid and resent to another client. Since each level of hierarchy is recreating the workunits from its producers for distribution, the deadline of the original workunit at the top level is not propagated. The problem is requesting too many or too few workunits in the hierarchy. In the first case the clients, may be normal or hierarchical, won't be able to upload them before the deadline passes, in the latter some of the clients are left without work. Predicting the performance is not the subject of this paper, but we needed a simple way to do it. We created a monitoring and statistics tool, which monitors the performance, number of users, hosts, sent and unsent workunits and many more. Since our main focus is on the *Local Desktop Grid* environment, where the performance should be less fluctuating, this enables us to have a good enough guess on the number of workunits to be requested based on the recent events.

Trust

BOINC uses an asymmetric key pair for code and workunit signing. When multiple clients interact with a single project the key pair is sufficient for authentication and authorization. In case where multiple projects interact with

each other additional information is required. We think it is also important to identify the origin of the application the project is currently using, since it may be from anywhere. This and the problems we describe in the following two sections can be easy solved by introducing *certificates* and defining various *administrative units*. We were considering GnuPG[16] and X.509[17]. GnuPG has a good infrastructure for key-distribution, but since X.509 is widely used and de facto standard for authentication and authorization[12, 14] we think it will provide a better solution for us.

Currently BOINC has one administrative unit, the *project* itself with it's key pairs. It's the task of the *project* to sign any application deployed and to sign the workunits sent to clients (using another key pair).

We want to distinguish between the *Application Developer*, the *Project*, the *Server* and the *Client*, each with a certificate assigned. The *Application Developer* is an individual or group of individuals who develop and sign their applications but she is often not involved in the management of the BOINC project. We think the application itself should not be a separate administrative unit, it can be identified among others with the signature of it's developer. The *Server* is the node hosting one or more *Project*. The *Client* is the *BOINC Local Client* or *CoreClient* whose task is to run the application of the *Project* with the given set of input data (*Workunit*).

Application representation

BOINC currently identifies the applications with a *name* and a *version*. This does not provide information about the developer or the origin of the application. We want to distribute them in the hierarchy, this requires a unique identifier for each version of each application. We want to extend the definition of the application by bundling the new signature and for the validation required certificates with it. This allows us to uniquely distinguish any application with the combination of the *name*, *version* and it's *Application Developer*'s signature.

Application registration

With the introduction of certificates and *Application Developer*s the application signing by the projects is not needed anymore. Instead, the project now publishes the list of trusted *Application Developers*. Thus applications can be distributed automatically in the hierarchy, but with security considerations.

Figure 3 shows the flow of the registration. Communication is always performed via the HTTP over TLS protocol (HTTPS)[7]. First the *Hierarchical Client* running on the *consumer* will contact the *producer* with the highest priority identifying itself with one of the defined *platforms*. Since both Projects have a *certificate* assigned they will authenticate mutually, they also have a list of the certificates of trusted *Projects* and *Application Developers*. This allows
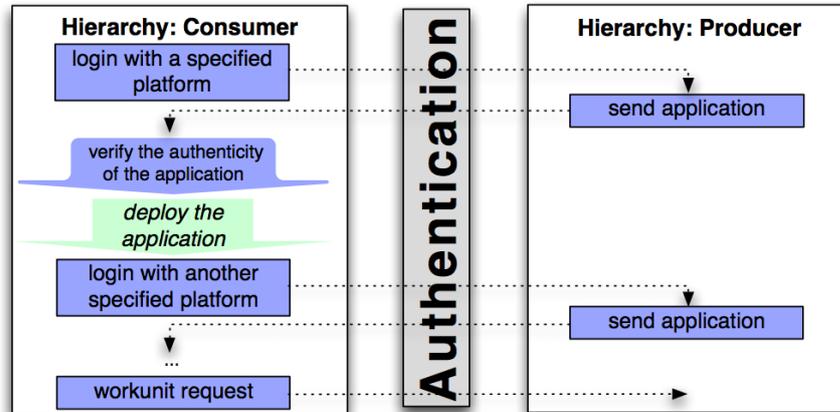
Figure 3. Application registration

to perform authorization on both sides. After a successful authentication and authorization the consumer will request the latest version of the application of the producer. The producer sends the executable, the signature and any certificates needed to verify the signature. The consumer verifies the authenticity of the application and check if its developer is authorized to deploy applications to the consumer Project. If authorized then the normal BOINC application deployment follows: copying the files to an HTTPS accessible directory and registering the metadata in the backend database. Updating and signing the list of trusted certificates is the responsibility of the project administrator.

The *Hierarchical Client* repeats this procedure for each platform defined. After querying all platforms, since no new application version is available it will start to request workunits.

The steps involved in the application registration process are presented in *Figure 4*. All communication between projects and clients is via the HTTP over TLS protocol. First step is for the *Application Developer* to sign her *Application*, producing a signature (*Sig*). Second step is to *Install* the application to *Project 2*. This initial installation is the task of the project administrator and is done manually. The project administrator adds the certificates required to verify the signature to the projects list of certificates (*Cert List P2*), runs the BOINC application registration procedure by copying the signature and executable(s) to the desired place and registers the metadata in the backend database. The project may sign the application, thus certifying its origin. When a *consumer* (*Project 1*) runs out of work the *Hierarchical Client* belonging to the project contacts a *producer* (Project 2). After mutual authentication the consumer downloads
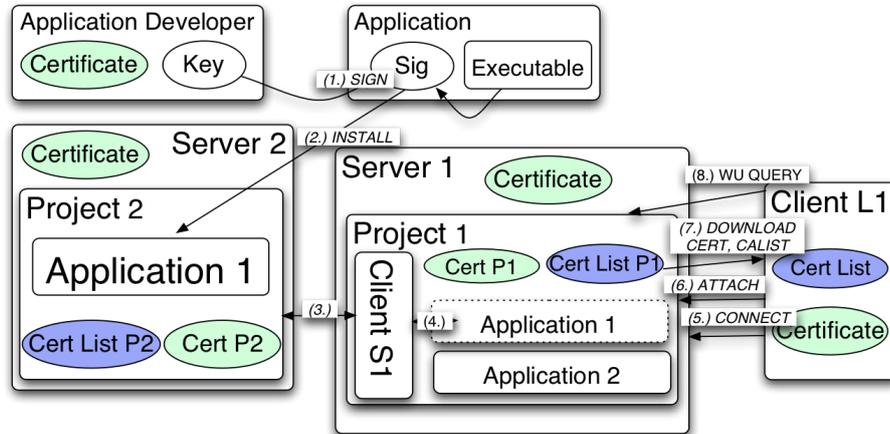
Figure 4.   Application registration and work distribution

Application 1, verifies the signature, authorizes the developer and *register*s the
application (*3.* and *4. step*). When the registration succeeds, the *Hierarchical
Client* will start requesting workunits from the producer and injecting them
in the consumer project. A *Local Client* contacts *Project 2* by connecting to
the server first (*5. step*). After mutual authentication the client is authorized
either by a certificate belonging to the *Client* or by a BOINC account key (*6.
step*). In the first case the project's list of certificates (*Cert List P1*) should
contain the client's certificate, this is another task which is performed manually
by the project administrator. The client downloads the application, and adds
all certificates from the project to its list (*Cert List*, *7. step*) and verifies the
application. The last step for the client is the downloading of workunits (*8.*).
After computing the result of a workunit, it is uploaded to *Project 1* where the
Hierarchical Client notices it and uploads it to the producer of the project and
reports it as finished.

5.   Conclusion and future work

SZTAKI Local Desktop Grid is based on BOINC, its main enhancement is
allowing hierarchical setups. Hierarchy allows to build larger desktop grids
by using existing projects as building elements. We have shown that it is
possible to have basic functionality for work distribution without modifying
already deployed projects. Our prototype implementation was tried in a test
environment revealing issues we need to address. We think security is crucial
for real-world deployment, and this increased security can be achieved by using

already proven technologies, like X.509 certificates. With the introduction of certificates, issues like application representation, distribution and registration can be solved. In the future we will work on solving the remaining issues and refining the security enhancements discussed. We want to work on estimating the number of required workunits to be transfered between different levels of the hierarchy. We also need to implement a better certificate distribution solution. Currently we don't have any certificate revocation implementation, so we want to address this problem in the future too.

6.   Acknowledgment

References

[1] Berkeley Open Infrastructure For Network Computing. http://boinc.berkeley.edu

[2] D. P. Anderson: BOINC: A System for Public-Resource Computing and Storage. 5th IEEE/ACM International Workshop on Grid Computing, November 8, 2004, Pittsburgh, USA.

[3] SETI@home: Search for Extraterrestrial Intelligence at Home. http://setiathome.berkeley.edu

[4] Peter Kacsuk, Norbert Podhorszki and Tamas Kiss. Scalable Desktop Grid System. Technical report, TR-0006, Institute on System Architecture, CoreGRID - Network of Excellence, May 2005.

[5] Jakob Gregor Pedersen & Christian Ulrik Sottrup. *Developing Distributed Computing Solutions Combining Grid Computing and Public Computing*. Master's thesis from University of Copenhagen, 2005.

[6] SZTAKI Desktop Grid. http://szdg.lpds.sztaki.hu/szdg.

[7] HTTP Over TLS. http://www.ietf.org/rfc/rfc2818.txt

[8] Sun Microsystems, JXTA. http://www.jxta.org.

[9] Distributed.net, The fastest computer on earth. http://www.distributed.net.

[10] Entropia, Inc. http://www.entropia.com.

[11] I. Foster, The Grid: Blueprint For a New Computing Infrastructure, Morgan Kaufmann, Los Altos, CA, 1998.

[12] I. Foster, C. Kesselman, S. Tuecke, The anatomy of the grid: Enabling scalable virtual organizations, Internat. J. Supercomput. Appl. 15 (3) (2001) 200-222.

[13] A. Grimshaw, W. Wulf, The legion vision of a worldwide virtual computer, Comm. ACM 40, 1997, 39-45.

[14] Foster, I., Kesselman, C., Tsudik, G. and Tuecke, S. A Security Architecture for Computational Grids. In ACM Conference on Computers and Security, 83-91.

[15] Myers, D.S., and M. P. Cummings. Necessity is the mother of invention: a simple grid computing system using commodity tools. Journal of Parallel and Distributed Computing, Volume 63, Issue 5, May 2003, pp. 578-589.

[16] The GNU Privacy Guard. http://www.gnupg.org

[17] Internet X.509 Public Key Infrastructure Certificate and CRL Profile. http://www.ietf.org/rfc/ rfc2459.txt