SZTAKI desktop grid: building a scalable, secure platform for desktop grid computing.

Attila Csaba Marosi[1]
Gábor Gombás[1]
Zoltán Balaton[1]
Péter Kacsuk[1]
Tamás Kiss [2]

[1] MTA SZTAKI, Computer and Automation Research Institute of the Hungarian Academy of Sciences
[2] School of Informatics, University of Westminster

This is a reproduction of CoreGRID Technical Report Number TR-0100, 28 August 2007, and is reprinted here with permission.

The report is available on the CoreGRID website, at:

http://www.coregrid.net/mambo/images/stories/TechnicalReports/tr-0100.pdf

---

SZTAKI Desktop Grid: Building a scalable, secure platform for Desktop Grid Computing

Attila Csaba Marosi, Gábor Gombás,
Zoltán Balaton, Péter Kacsuk
{atisu, gombasg,
balaton, kacsuk}@sztaki.hu

MTA SZTAKI, Computer and Automation Research Institute
of the Hungarian Academy of Sciences
H-1528 Budapest, P.O.Box 63, Hungary

Tamás Kiss
kisst@wmin.ac.uk

Centre for Parallel Computing, University of Westminster
115 New Cavendish Street, London W1W 6UW, UK

CoreGRID Technical Report
Number TR-0100
August 28, 2007

Institute on Architectural Issues: Scalability, Dependability, Adaptability

CoreGRID - Network of Excellence
URL: http://www.coregrid.net

CoreGRID is a Network of Excellence funded by the European Commission under the Sixth Framework Programme

Project no. FP6-004265

SZTAKI Desktop Grid: Building a scalable, secure platform for Desktop Grid Computing

Attila Csaba Marosi, Gábor Gombás,
Zoltán Balaton, Péter Kacsuk
{atisu, gombasg,
balaton, kacsuk}@sztaki.hu

MTA SZTAKI, Computer and Automation Research Institute
of the Hungarian Academy of Sciences
H-1528 Budapest, P.O.Box 63, Hungary

Tamás Kiss
kisst@wmin.ac.uk

Centre for Parallel Computing, University of Westminster
115 New Cavendish Street, London W1W 6UW, UK

CoreGRID TR-0100

August 28, 2007

Abstract

The Desktop Grid model harvests unused CPU cycles of connected computers. In this report we present a concept how separate 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 for enhanced security to satisfy the requirements for real-world deployment.

1 Introduction

Contrary to traditional grid [1] 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.

The common architecture of desktop grids typically 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 tasks of an application with a set of input data. When the task has finished, the client uploads the results to the server where the application assembles the final output from the results returned by clients.

A major advantage of desktop grids over traditional grid systems is that they are able to utilize non-dedicated machines. Besides, the requirements for providing resources to a desktop grid are very low compared to traditional grid systems using a complex middleware. Thus, a huge amount of resources can be gathered that were not available for traditional grid computing previously. Even though the barrier may be low for resource providers, deploying a

This research work is carried out under the FP6 Network of Excellence CoreGRID funded by the European Commission (Contract IST-2002-004265).

desktop grid server is a more challenging task because a central server creates a central point of failure and a potential bottleneck, while replicating servers requires more efforts.

Users of scientific applications are usually only concerned about the amount of computing power they can get and not about the details how a grid system delivers this computing power. Unfortunately existing applications may have to be modified in order to run on desktop grid systems which makes desktop grids less attractive for application developers than traditional grid systems.

Based on the environment where the desktop grid is deployed we can distinguish between two different desktop grid flavors.

Global Desktop Grids Global Desktop Grids (also known as Public Desktop Grids or Public Resource Computing) consist of a server which is publicly accessible over the Internet, and the attached clients are offered by their owners to help out projects they sympathize with. 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 which started a computation will indeed finish it. Furthermore, the clients cannot be trusted to be free of either hardware or software defects or malicious intent, 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, such as for medical applications, this is even forbidden by law). Thus, in a Local Desktop Grid the project and clients are usually shielded from the world by firewalls or other means and only known and trusted clients are allowed to offer their resources. This environment gives more flexibility by allowing the clients to access local resources securely and since the resources are not voluntarily offered the performance may be limited but more predictable. However, new security requirements arise in Local Desktop Grids that require authentication of clients and servers and establishing trust between parties.

The rest of the report is organized as follows. The next section introduces SZTAKI Desktop Grid. In Section 3 we describe our extension of BOINC to be able to support hierarchy. In section 4 we describe an enhanced BOINC security model for hierarchy and how to further extend the model for industrial use. We present the current challenges, limitations and our proposed solutions in Section 5. Then the conclusion section closes the report.

2 SZTAKI 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 [3] (or SZTAKI LDG) implements the latter. It is based on BOINC [4][16] (which aims to provide an open infrastructure for Public Resource Computing) and is aimed to satisfy the needs of both academical institutions and enterprises. But what if there are several departments using their own resources independently and there is a project at a higher organizational level (e.g. at a campus or enterprise level)? Ideally, this project would be able to use free resources from all departments. However, using BOINC this would require individuals providing resources to manually register to the higher level project which is a high administrative overhead and it is against the centrally managed nature of IT infrastructure within an enterprise.

As others before us, we faced several possibilities when designing SZTAKI LDG: to develop our own solution [5], to use other desktop grid systems and approaches like Distributed.net [15], Legion [6], JXTA [18], Entropia [7] or XtremWeb [8]. We decided to build on BOINC, because it is a proved technology, has a large user base [9], its open source, cross-platform and has a clean design and implementation making it the best target for third-party enhancements [2]. SZTAKI Desktop Grid also has a Public Desktop Grid version [17] running currently with more than 17000 registered users.

3 Hierarchy of Desktop Grids

One of the enhancements of the SZTAKI Local Desktop Grid is hierarchy [10]. It 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. The hierarchical desktop grid allows a set of projects to be connected to form a directed acyclic graph. Work is distributed among the edges of the directed graph. The projects are ordered into levels based on the distance between them and the top level.

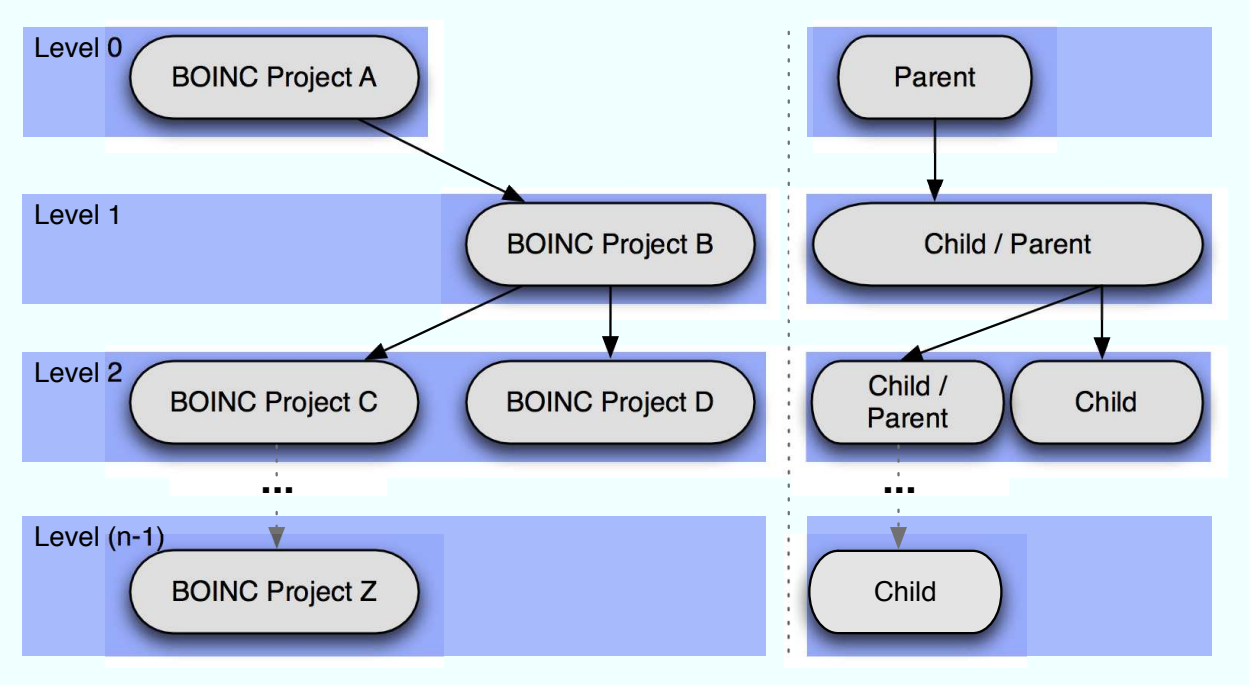

Figure 1: Roles in the hierarchy

Every project has a classical parent-child relationship with the others. A project may request work from a project above (*child*) or may provide work for a project below (*parent*). The hierarchical interaction is always between a parent and a child regardless of how many levels of hierarchy are above or below them. For a child every workunit is originating from its parent regardless where it is originally from or from where was the input data for the workunit fetched (although the data is not always from the parent). It is allowed for a project to have more children and parents. Figure 2 shows a three-level example.

The Hierarchy Client, which is a modified BOINC Core Client, is always running beside the child project. Thus at the top level there is no need for any modifications, it is just an ordinary BOINC project. Generally, a project acting as a parent does not have to be aware of the hierarchy, it only sees the child as one powerful client. The client reports to the parent a pre-configured number of processors, thus allowing to download the desired number of workunits. There can be limitations set on the server side to maximize the allowed number of workunits downloaded per client, so the only requirement for the parent side is to set these limits sufficiently high.

The Hierarchy Client has two components (see Figure 2): a master side which puts retrieved workunits in the database of the LDG and retrieves the completed results, and a client side which downloads workunits from the parent and uploads results.

Using a prototype with this functionality we were able to provide basic hierarchical functionality without any other modifications, but it had several drawbacks:

- the application binaries had to be deployed manually on each level.
- since workunits refer to an application by its name and version for execution, there is no guarantee that there won't be name collisions between new and already deployed applications when there are a large number of applications deployed in the hierarchy.

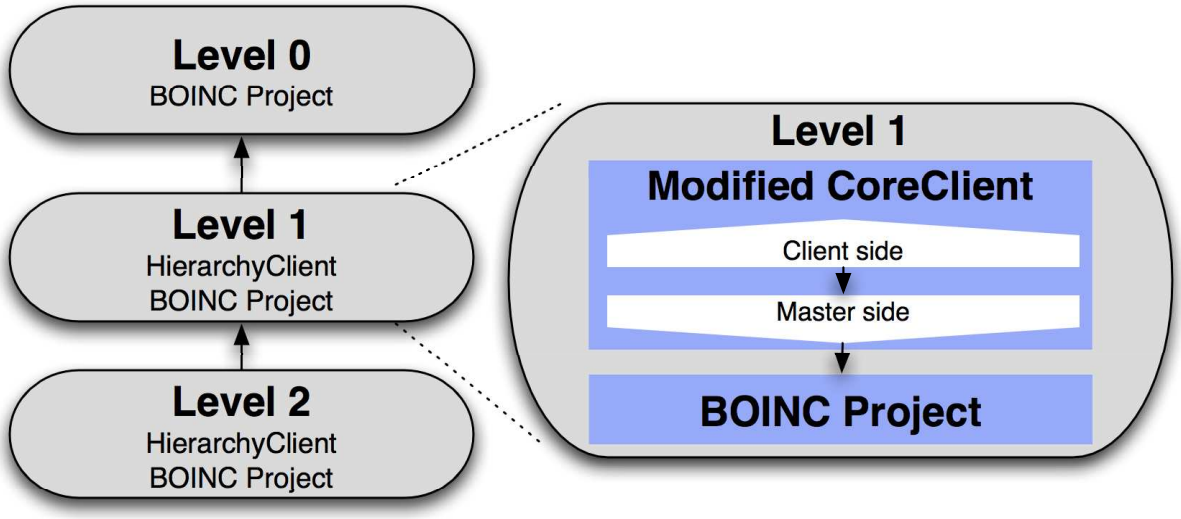

Figure 2: The split architecture of the Hierarchy prototype. Inside the Core Client the Client side is acting as a child by requesting work and the Master side as a parent by providing work for the project.

- work distribution is based on the local scheduling [12] method implemented in the BOINC Core Client which is not ideal in a hierarchical setup as it was not designed for this task.

These limitations need to be addressed in order to provide a fully-working model. In the next section we describe how did we extend BOINC to achieve the required functionality.

4 Extending BOINC for Use in Hierarchy

Although the hierarchy prototype presented in the previous section is very simple and was easy to implement, it had a major drawback: applications must be installed manually at every child level in order to be able to process workunits originating from the parent. Overcoming this limitation also requires replacing of the security model of BOINC.

The most important factor in desktop grid computing is the trust between the clients and the project providing the application. Allowing foreign code to run on a computer always has a risk of either accidental or intended misbehavior. BOINC mitigates this risk by only allowing to run code that has been digitally signed by the project the client is connected to. Clients trust the operators of the BOINC project not to offer malicious code, and digitally signing the application provides technical means to ensure this trust relation.

Of course it is not enough to only sign the application binary, the input data must be signed as well (think of the case when the application is some kind of interpreter and the input data can instruct it to do just about anything). Therefore BOINC uses two separate key pairs: one is used to sign the workunits (which in this context means the set of input files and a link to the application binary), the other is used to sign the application code. The private key used for workunit signing is usually present on the project's central server, while the private key used for application signing is usually kept at a separate location. The different handling of the private keys stems from their usage pattern: the workunit signing key is used very often while the code signing key is seldom needed therefore it can be protected better. This technique significantly reduces the risk of compromising the application signing key even if the machine hosting the project is compromised, but this also means that installing new applications is a manual process – which is unfortunate for a hierarchical setup.

Therefore, solving the automatic application deployment issue presents two challenges:

- a lower-level project in a hierarchical desktop grid system must be able to automatically obtain an application's binary from its parent and be able to offer the application to its clients without manual intervention, and
- this process must not increase the risk of injecting untrusted applications into the system.

These requirements mean that a lower-level project can not simply re-sign the application it has obtained from the parent, since that would require the private key to be accessible on the machine hosting the lower-level project which in turn would significantly increase the risk of a key compromise if the machine hosting the project is compromised.

4.1 Extending the Security Model to Support Hierarchy

As discussed above the security model used by BOINC is not adequate in a hierarchical setup and a new model is needed. The model must provide enough information for the operator of the client machine ($User$ from now on) to decide if a downloaded workunit should be trusted to run on the client machine or not, independent from where in the hierarchy the workunit is originated from. The model must provide enough information for the following decision scenarios:

1. The $User$ wants to trust any workunits of applications installed locally on the BOINC project she is directly connected to (i.e., the $User$ trusts the project itself). This is the original trust model of BOINC.
2. The $User$ wants to trust any workunits from a given project, regardless of how many levels of hierarchy did the workunit travel through. This is in fact a generalization of the previous requirement.
3. The $User$ wants to trust a specific application regardless of where in the hierarchy it is hosted and regardless of what other applications does the hosting project offer.

The $t(\langle subject \rangle, \langle object \rangle)$ trust relation for a workunit can be broken down to three parts:

- trusting the application code: $t(User, App)$,
- trusting the set of input files: $t(User, Input)$, and
- trusting the link between the application, its inputs and the desired location of its outputs to prevent the application from processing data that was meant for an other application: $t(User, \langle App, Input, Output \rangle)$. We will use the shorthand $WUDesc$ for the $\langle App, Input, Output \rangle$ triplet.

A workunit WU is trusted if all components are trusted: $t(User, App) \wedge t(User, Input) \wedge t(User, WUDesc) \rightarrow t(User, WU)$.

The trust relation is realized by digital signature verification. Therefore, each of the three classes of objects App, $Input$ and $WUDesc$ are accompanied by one or more digital signatures $Sig_X : X \in \{App, Input, WUDesc\}$, and it is assumed that $User$ has a set of trusted identities marked $TrustedID_{User}$. Thus the trust relation becomes $t(User, X) \iff \exists s \in Sig_X : verify\_sig(X, s) \wedge subject\_of(s) \in TrustedID_{User}$, where the $subject\_of(s)$ function provides the identity that created the signature s. We also allow special $Any_X : X \in \{App, Input, WUDesc\}$ elements which satisfy the $\forall s : verify\_sig(Any_X, s) = TRUE$. $Any_X \in TrustedID_X$ means that the user does not require a valid signature for that particular component.

We decided to use the X.509 Public Key Infrastructure, since it is a widely accepted and used infrastructure that provides all the technical elements we need. Therefore, the $TrustedID_{User}$ set becomes a list of X.509 certificates.

We define 3 entities responsible for signing various components of the system. The Application Developer ($AppDev$ from now on) can sign application code. This kind of signature testifies that the application binary comes from a known source and does not contain malicious code. The $Project$ is the administrative body of the BOINC project and it may also sign application code testifying that said application is in fact part of the project. The $Server$ is the machine where the project is hosted, and it signs input files and workunit descriptors. Using the original BOINC terms the $AppDev$ provides the code-signing key, while the $Server$ provides the workunit-signing key.

The $TrustedID_{User}$ list of trusted certificates must be determined by the user, since the trust is ultimately a human relation. This may be simplified by the $Project$ by providing a list of $Server$ and optionally $AppDev$ certificates it trusts – this means the user can delegate the trust to the $Project$. This realizes the first scenario described in 4.1. The second scenario is realized if the $Project$ also provides the aggregated list of certificates from all levels above it in the hierarchy. The third scenario is realized if the user lists only the certificate of the appropriate $AppDev$ and specifies that she does not care about the signature of $Input$ or $WUDesc$.

4.2 Extending the Security Model for Industrial Needs

The previous section described a model how a user can trust work received from a hierarchical desktop grid system. In an industrial environment however more is needed: it is not enough for the user to trust the workunit, but the project must also trust the user before it gives out possibly confidential information. Also it is not enough just to trust the receiving user, but the data also has to be protected from being disclosed to untrusted parties. This is a new requirement that is not present in public projects.

Protecting the confidentiality of the data can be easily achieved. BOINC by default uses plain HTTP protocol for communication, but it also supports the HTTPS protocol where the communication is encrypted. The $Server$ certificate can be used with the HTTPS protocol to ensure that the $User$ in fact talks to the server she thinks is talking to. Although BOINC uses a simple shared-secret based authentication scheme to identify users, this authentication applies only to interactions with the scheduler. Together with the use of HTTPS this may be adequate to prevent unauthorized users from uploading results, but it does not prevent unauthorized users to download application code and input data if they are able to guess the file name used on the server.

The protection of input data from unauthorized download can be achieved by giving every user a certificate. The $Project$ can act as a Certificate Authority and can sign the certificates of all authorized users. Then, the web server that is used for downloading the input files can be configured to only allow downloading if the client authenticated itself with a properly signed certificate.

The workunits are always signed by the server running a specific project, so the projects need a way to make their known and accepted signing certificates available for their clients and other projects. This is solved by an extension to the web based interface of the BOINC project allowing to query for the certificates via the HTTP(S) protocol and depending on the trust model described in 4.1. Although it is a simple extension on the server side the BOINC Core Client needs modifications to be able to query for certificates.

4.3 Automatic Application Deployment

BOINC allows the creation of a workunit that refers to external servers for the input files. This means that lower-level projects in a hierarchy do not need to install the input files locally, they may just refer to the original location of the files in the workunit description. However due to security considerations BOINC does not allow to refer to outside of the project for application binaries, they must always reside on the project's server. Thus, lower-level (child) projects must deploy all applications whose workunits they offer locally.

The automatic deployment of applications presents two problems. The first problem arises from the need to properly sign the binary and is solved by the introduction of the $AppDev$ role as described in the previous section. If the users have configured their $TrustedID_{User}$ sets to contain the appropriate certificate of the $AppDev$, then the project does not need to sign the application binary, thus its secret key is not needed for application deployment.

The second problem arises from the fact that BOINC uses the $\langle AppName, Version \rangle$ tuple to identify applications and in a complex hierarchy it is possible that at different levels different applications are installed under the same name. This problem can be solved by automatically renaming the application when a workunit is transferred from a parent to lower level child project. Using an Universally Unique Identifier (UUID) as the new application name ensures that there will be no name collisions.

For the following we assume that the application consists of just a single binary. Compound applications or applications with accompanying shared libraries are not considered in this report.

The hierarchy client keeps track of the name mapping of the application between parent projects and child project. Such a renaming is possible because on the sever side only the workunit-generating master application cares about the name of the application, and in this case this master application is the link between the members of the hierarchy and therefore has full control. The UUID is generated by the hierarchy after downloading it from the parent project, before registering at the child project. Additionally, the following requirements have to be met for the application registration in a Hierarchical Desktop Grid:

- The registration method should be consistent with the original registration method, allowing already deployed projects to be added to a hierarchy without any modification and any project to leave the hierarchy anytime.
- Different versions of the same application should be allowed to run in parallel, since each parent may run different version of the same application.

- Since each application instance is tied to a platform, the application name should be the same for all platforms, allowing any child to query for the different platform instances of the application.
- Instances of the same application originating from different parents should be treated as different ones, to ensure that results are reported to the appropriate parent.

The flow of the deployment is the following.

1. The Hierarchy Client periodically queries higher level projects for new applications. When a new application is available it receives the $\langle App, AppName, Version, Signatures \rangle$ tuple identifying the application for a given $Platform$.
2. The $Signatures$ are checked against the $TrustedID_{Project}$ set of the child project containing all accepted $AppDev$ and $Project$ certificates.
3. The $\langle AppName, Version, Signatures \rangle$ triplet is checked against the list of applications already registered for a specific parent.
   a. If found, the application is already available at the child project.
   b. If not found, the Hierarchy Client creates a new mapping: $\langle AppName, Version, Signatures, Parent \rangle \rightarrow \langle UUID, 1.0 \rangle$
4. The Hierarchy Client registers the application with BOINC using $UUID$ as the application name and 1.0 as application version.

The above procedure ensure that applications can still be installed manually as in a regular BOINC project and that will not cause inconsistency between the configuration files of the project, the database of the project and the Hierarchy Client. There is one significant difference though: an automatically deployed application is not signed using the code-signing key of BOINC, instead the signature retrieved by the Hierarchy Client is used. This requires that the Core Client requesting work (and receiving applications) is able to retrieve the certificates (depending on the trust scenario described in 4.1) from the given project, and is able to validate the signature of the application (and the ones of the workunits belonging to it) using the certificates.

4.4 Application Deployment and Work Distribution

This section gives an overview of the application and work distribution in the Hierarchical Desktop Grid. In our example scenario we use the simplest setup, which consists of just two projects $Project\ A$ and $Project\ B$, one application App and one $User$. The flow of the deployment and distribution process is the following:

1. The application developer $AppDev$ may initially sign the App using her secret key.
2. The certificate of the $AppDev$ may be added, if not already done so, to the list of certificates belonging to $Project\ A$ where the application is about to be installed by the administrator of the project.

   The list of certificates belonging to an entity (server, project, or client) holds all the certificates of the application developers, projects, servers and clients accepted by the entity.
3. The application is installed by the administrator manually. This initial procedure is the same as the normal application install process of BOINC.
4. The $Project$ may also sign the application. This signature may either be appended to the signature of the $AppDev$ or it may replace the original signature if the project does not wish to disclose the origin of the application. This step must be performed manually since the secret key of the $Project$ should not be kept on the same machine where BOINC is running.
5. Workunits are created by the master application and are passed to BOINC.

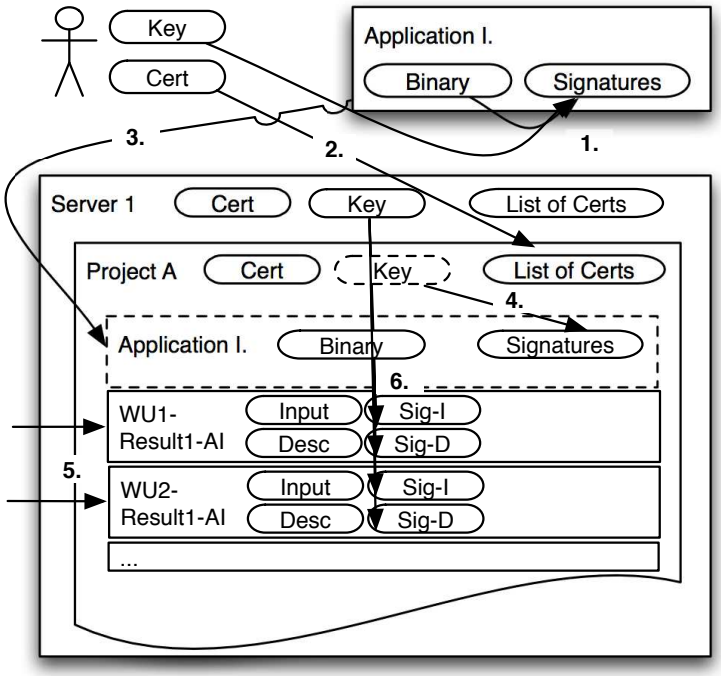

Figure 3: Application deployment and work distribution

6. For each workunit the input data ($Input$) and workunit descriptions ($Desc$) are signed by the $Server\ 1$ ($Sig-I$, $Sig-D$).

At this point the results are ready to be sent to any client attached to the project. Clients may be normal BOINC Core Clients or Hierarchy Clients.

7. The $Hierarchy\ Client$ connects. $Server\ 1$ has a list of the certificates of all accepted clients. If the certificate of the $Hierarchy\ Client$ is among them, it can continue to attach to the desired project running on the server. The project has a list of certificates too, containing the certificates of the accepted clients.

8. The $Hierarchy\ Client$ checks for new applications. Each application is tied to a BOINC platform (OS and architecture combination). The $Hierarchy\ Client$ will query for applications tied to each predefined platform. The application binary and the belonging signatures are downloaded.

9. The signatures of the application binary are verified using the client's list of certificates. $User$s have a $TrustedID_{User}$ set defined, but the $Hierarchy\ Client$ delegates the trust to the child project, in this case to $Project\ B$. It will accept any application $Project\ B$ is trusting.

10. A unique name for the application is created, and the $Hierarchy\ Client$ stores the name mapping as described in 4.3. The unique name guarantees that there will be no name collisions in the hierarchy, and the mapping allows the $Hierarchy\ Client$ to update/remove applications at the child project. $Project\ B$ might add its signature to the application, certifying the path of origin for its children.

At this point the application is deployed at the child project with the unique name. $Hierarchy\ Client$ will continue querying for new applications (checking all available platforms) and repeat this procedure (8-10) until there are no new ones available.

11. The $Hierarchy\ Client$ will now query for work for the applications deployed at $Project\ B$. The name mapping is used in this process, since for the same application a different name is set at the child and at the parent. A

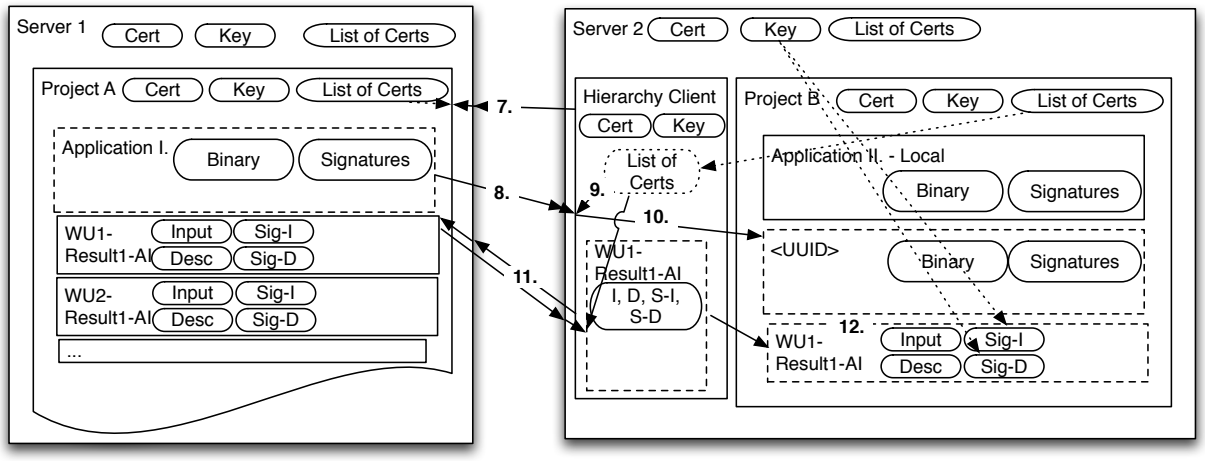

Figure 4: Application deployment and work distribution

successful query will fetch a result, which consists of one or more input files, their signatures, and a workunit description (it is the same for each result created from the same workunit) and its signature.

The signatures of the input files and workunit descriptions ($Sig - I$, $Sig - D$) are checked against the $TrustedID_{Project}$ set of the child project.

12. From the result fetched from the parent a workunit is created at the child project by the $Hierarchy\ Client$. $Server\ 2$ may add its signature to the inputs and descriptions belonging to the newly created workunit. From the workunit one or more results ($WU1 - Result1 - AI$) are created by the child project.

At this point the application ($< UUID >$) and a workunit belonging to it is fully deployed at the child project, waiting to be downloaded by a client, which may be a Hierarchy Client or a Core Client. If a Hierarchy Client connects, the procedure is the same from step 7, if a $User$ (using her Core Client) connects the following steps will be executed.

13. A $User$ connects to the server. $Server\ 2$ and $Project\ B$ has all the certificates of the accepted clients pre-installed, meaning they can authenticate her. Afterward her $Client$ queries for new applications belonging to its platform, and downloads their binary and signatures.

14. The signature belonging to the downloaded application is verified that it is by one of the trusted application developers, and if there are additional signatures, they are verified that they are by one of the trusted projects.

15. The $Client$ will now query for work (results) belonging to one of the applications available at the client (the application is chosen by the local scheduling implemented in the Core Client). On success one or more results ($WU1 - Result1 - AI$) consisting of input files ($Input$), workunit description ($Desc$) and their signatures ($Sig - I$, $Sig - D$) will be downloaded. The signature(s) of the description and the input files are verified to ensure they are signed by (one of) the trusted servers.

16. The result $WU1 - Result1 - AI$ is ready to be processed by the application. Processing it, will produce one or more output files ($Output$). The $Client$ signs these files.

17. The output files and signatures ($Output$, $Sig - O$) are uploaded to $Project\ B$ by the $Client$, and the result is reported as finished.

18. The signatures of the uploaded files are checked if they are created by one of the trusted clients, using the list of certificates of $Project\ B$.

19. The $Hierarchy\ Client$ notices that a result belonging to a workunit that was created by it is complete. It fetches the output files from $Project\ B$, so it is able to upload it to the parent when needed. It adds its signature to the

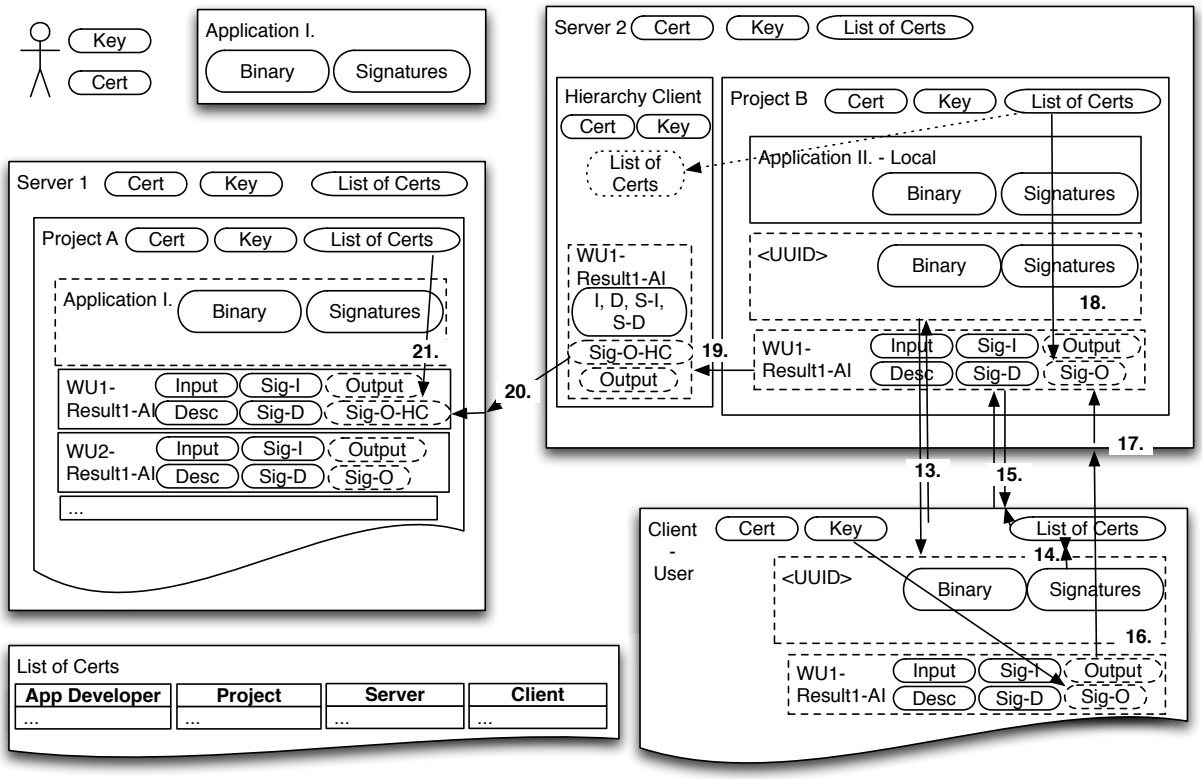

Figure 5: Application deployment and work distribution

output(s) of the result ($Sig - O - HC$). For the parent project the $Hierarchy\ Client$ is the $Client$ processing work, but in reality it is acting as a middle-man relaying work and binaries between the two projects.

20. The $Hierarchy\ Client$ contacts $Server\ 1$, $Project\ A$, and uploads the output files and their signatures belonging to the result.

21. The $Output$ is verified using the signature and the list of certificates by $Project\ A$.

At this point the completed result is available at $Project\ A$ for validation. Workunit validation is performed only here at the originating project, the child projects use a trivial validator which is part of the $Hierarchy\ Client$, and it is accepting all incoming results. This may be adequate in a controlled environment, where only the selected clients allowed to return results, but this does not filter out syntactically incorrect results at the lower levels caused for example by some hardware defect.

5 Future Work

Our enhancements improve the original security model of BOINC in many ways, and the Hierarchical Desktop Grid allows to gather the resources of any hierarchical structured organization with less management overhead, but there are several limitations we are aware of. In the following subsections we discuss these limitations and introduce our proposed solutions.

5.1 Sandboxing

Another aspect of security that we did not mention yet is isolating the application from the rest of the computer it is running on. The BOINC Core Client simply *fork*s a new process for each application it is executing, meaning that the

application process has access to the same resources as the Core Client itself. In an industrial environment sometimes the data on the computer (confidential information) is needed to be shielded off from the application code run by the client. To achieve this the Core Client may be run as a restricted user which also restrict the processes created by it, but in industrial environments the platform used is often Windows and it is sometimes not enough to only rely on the operating system facilities to ensure isolation from the rest of the system. In a UNIX environment the sandboxing can be easily achieved, since there are several tools like XEN [13] or *chroot* available. Unfortunately these tools are not available for Windows. According to our present knowledge there is no other similar mechanism for widely used versions of Windows (2000, 2003 or XP) either. A possible solution would be using virtualization technologies available for all platforms like VMware [21], VirtualBox [22], Bochs [19] or QEMU [20].

We propose that instead the simple *fork* mechanism a lightweight virtual machine with a minimalist Linux image should be started with a virtual machine monitor like QEMU. This would properly isolate the application from the rest of the computer of the User. Also because the virtual machine runs Linux independent of the operating system on the User's computer this way only a version of the application for the Linux platform would be required that simplifies application development.

5.2 Redundancy

Redundancy in BOINC increases the probability that 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.

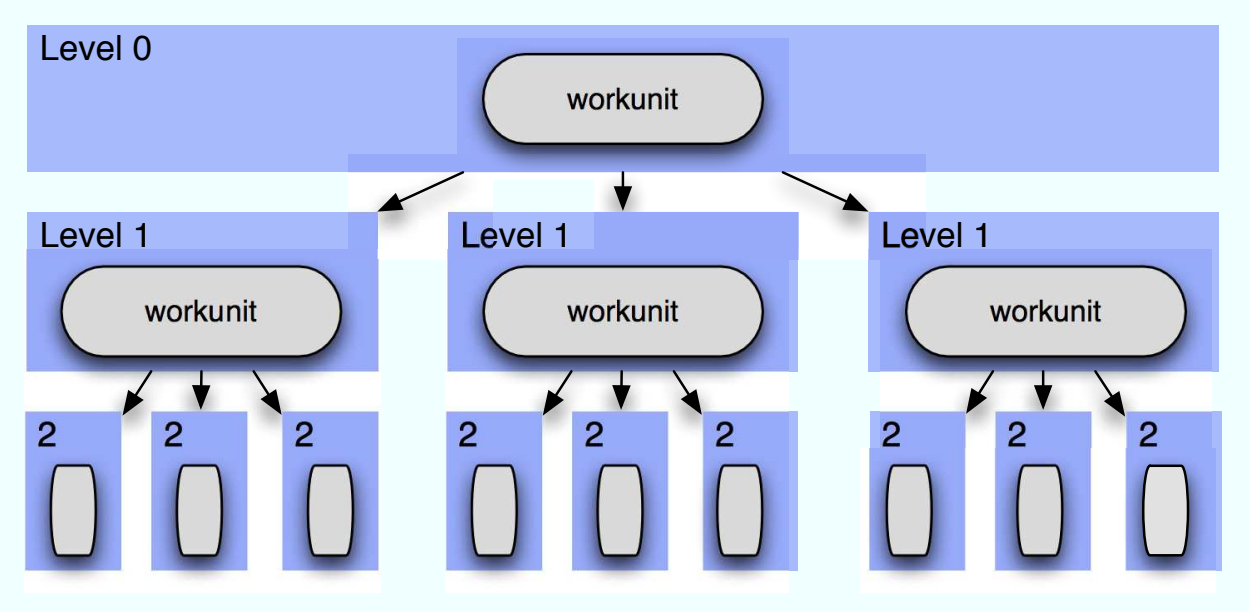

Figure 6: Growing number of redundant workunits in the hierarchy demonstrated with a simple three level layout.

Figure 6. shows a three level layout with the redundancy of three on each level. In this case each parent on each level creates three copies of any workunit received. By the second 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. This problem can be solved by forcing redundancy to be disabled on all but the top level. This way exactly the requested number of redundant workunits will be distributed.

5.3 Scheduling

The Hierarchy Client currently uses the scheduling method in the BOINC Core Client, which is intended for clients requesting work for themselves, not for hierarchical work distribution. Currently we are adjusting the number of

processors reported by the client to adjust the number of requested workunits.

Another problem comes from the fact that BOINC assigns a deadline to each downloaded workunit to prohibit workunit-hijacking by users. The deadline is set when the workunit is downloaded and after it passes, the workunit is considered invalid and resent to another client. The deadline is the sum of the time of download and a delay bound value. Since each level of hierarchy is recreating workunits from those it got from its parent for distribution, the deadline of the original workunit at the top level is not propagated. Thus the lower level projects have no information if their workunits will be invalidated on a higher level because the deadline has already passed. A solution would be to make the workunits carry the original deadline with them via their descriptors as they traverse the hierarchy. This would allow to give the lower level projects some idea how to set the delay bound value of their workunits upon registration. Ideally this value should be updated upon the download of the workunit, since only then is the time of download known, and so the deadline could be set exactly.

In a hierarchy there is the problem of requesting too many or too few workunits. In the first case the clients (that may be Core or Hierarchy Clients) won't be able to upload them before the deadline passes, in the latter case some of the clients are left without work.

Predicting the performance is not the subject of this report, but we needed a simple way to do it. Thus, we developed an own 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 will enable us to have a good enough guess on the number of workunits to be requested based on the recent events. For the long term we need to develop scheduling strategies specific for the Hierarchical Desktop Grid.

6 Conclusion

In this report we demonstrated how can stand-alone desktop grid installations be combined to form a large-scale grid system. We described our extensions for the security model that allows SZTAKI Desktop Grid to fulfill the additional security requirements that follow from the hierarchical setup and those required by industrial use cases. Future work includes working on the enhancements described in the previous section and various other tasks like improved certificate management and certificate revocation.

SZTAKI Local Desktop Grid (LDG) represents a matured Desktop Grid technology that can be used even in industrial environments. Components of the Hierarchical LDG will be shortly made available via the SZDG web page [17].

7 Acknowledgments

The research and development published in this report is partly supported by the Hungarian Government under grant NKFP2-00007/2005 (Development and Meteorological Application of New Generation Grid Technologies in the Environmental Protection and Building Energy Management Project) and by the European Commission under contract number IST-2002-004265 (FP6 NoE, CoreGRID).

References

[1] Ian Foster: *The Grid: Blueprint For a New Computing Infrastructure*, 1998

[2] Jakob Gregor Pedersen, Christian Ulrik Sottrup: *Developing Distributed Computing Solutions Combining Grid Computing and Public Computing*, M.Sc. from University of Copenhagen, 2005

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

[4] David P. Anderson: *BOINC: A System for Public-Resource Computing and Storage*. In proceedings of the Fifth IEEE/ACM International Workshop on Grid Computing, Pages 4-10., 2004.

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

[6] A. Grimshaw, W. Wulf: *The Legion vision of a worldwide virtual computer* In the Communications of the ACM, Volume 40, Pages 39-45

[7] Andrew A. Chien: *Architecture of a commercial enterprise desktop Grid: the Entropia system*, In Grid Computing: Making the Global Infrastructure a Reality, Chapter 12, Pages 337-350, 2003

[8] Franck Cappello, Samir Djilali, Gilles Fedak, Thomas Herault, Frederic Magniette, Vincent Neri, Oleg Lodygensky: *Computing on large-scale distributed systems: XtremWeb architecture, programming models, security, tests and convergence with grid*, Future Generation Computer Systems, Volume 21/3, Pages 417-437, 2005

[9] David P. Anderson, Gilles Fedak: *The Computational and Storage Potential of Volunteer Computing*, In Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid, Pages 73-80., 2006

[10] Attila Csaba Marosi, Gabor Gombas, Zoltan Balaton: *Secure application deployment in the hierarchical local desktop grid*. In Proc. of DAPSYS 2006 6th Austrian-Hungarian Workshop on Distributed and Parallel Systems, 2006. September

[11] Zoltan Balaton, Gabor Gombas, Peter Kacsuk, Adam Kornafeld, Attila Csaba Marosi, Gabor Vida, Norbert Podhorszki, Tamas Kiss: *SZTAKI Desktop Grid: a Modular and Scalable Way of Building Large Computing Grids*, Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International 26-30 March 2007

[12] David P. Anderson, John McLeod VII.:*Local Scheduling for Volunteer Computing*, Workshop on Large-Scale, Volatile Desktop Grids (PCGrid 2007), 2007

[13] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, A. Warfield: *Xen and the Art of Virtualization*. In Proceedings of the 19th ACM SOSP, pages 164-177, October 2003.

[14] RFC 2818: HTTP Over TLS. `http://www.ietf.org/rfc/rfc2818.txt`

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

[16] BOINC: Berkeley Open Infrastructure for Network Computing. `http://boinc.berkeley.edu/`

[17] SZTAKI Desktop Grid. `http://www.desktopgrid.hu/`

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

[19] Bochs: Think inside the bochs. `http://bochs.sourceforge.net/`

[20] QEMU: Open source processor emulator. `http://fabrice.bellard.free.fr/qemu/`

[21] VMware. `http://vmware.com/`

[22] VirtualBox. `http://www.virtualbox.org/`