TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

EDUARDO JAVIER CORTIJO SERRANO

VOLUNTEER COMPUTING ON DISTRIBUTED UNTRUSTED NODES

Master's thesis

Examiner: MSc Ville Seppänen and
Prof. Jarmo Harju

Examiner and topic approved by the Faculty Council of the Faculty of Computing and Electrical Engineering on 5th April 2014.

Abstract

TAMPERE UNIVERSITY OF TECHNOLOGY
CORTIJO SERRANO, EDUARDO JAVIER: Volunteer computing on distributed untrusted nodes
Master of Science Thesis, 55 pages, 1 Appendix pages
April 2014
Examiner: MSc Ville Seppänen, Prof. Jarmo Harju
Keywords: volunteer computing, distributed computing, crowdsourcing, build automation, trust, reputation management

The growth in size and complexity of new software systems has highlighted the need of more efficient and faster building tools. The current research relies on automation and parallelization of tasks dividing and grouping software systems in dependent software packages. Some modern building systems as Open Build Service (OBS) centralize sources commitment and dependencies solving for Linux distributions. After, they distribute these heavy build tasks among several build hosts, to finally deliver the results to the community.

The problem with these building services is that as they are usually supported by non-commercial communities, the resources to maintain the build hosts are less. Because of this, the idea of distributing these jobs among new building hosts owned by volunteers is tempting. However, carrying out this idea brings new challenges and problems to be solved, concerning the new pool of untrusted, unreliable workers.

This thesis studies how the concept of volunteer computing can be applied to software package building, specifically to OBS. In the first part, the existing platforms of volunteer computing are examined showing the current research and the pros and cons of using them for our purposes.

The research of this thesis led to a different solution called Volunteer Worker System (VWS). The main concept is to provide a centralized system that serves OBS reliable trusted workers compiling the results sent by the volunteers. Each worker acts as a proxy between the untrusted volunteers and the OBS server itself, validating by multiple cross-checking the results obtained. The volunteers from the volunteer pool are grouped to serve each surrogate depending on OBS needs.

A simple proof-of-concept of the designed system was set-up on a network distributed environment. A host acting as Volunteer System groups and dispatches jobs coming from a host simulating OBS server to several volunteer workers in separate hosts. These volunteers send back their results to the Volunteer System to validate and forward them to OBS Server.

Ensuring security on the designed solution is one of the needs to deploy the system on a real-environment. The OBS instance receiving the volunteers work needs to be sure that the Volunteer System offering them is fully trusted. Also, a whole front-end system to attract and maintain volunteers needs to be implemented.

Preface

First, I would like to thank the Tampere University of Technology and the Pervasive Computing department, specially to Prof. Jarmo Harju, for providing me the resources and the opportunity to work here in my Master's Thesis. Also, to Bill Silverajan for presenting and offering me the thesis topic I enjoyed working on.

Second, I want to thank my supervisor MSc Ville Seppänen for patiently guiding me through the process and for letting me incorporate my own ideas on the design.

Furthermore, I would like to thank my family for supporting me one more year and also to my friends, here and there, for making these months easier to me.

Tampere, April 16th 2014

Eduardo Javier Cortijo Serrano

Table of Contents

Abbreviations

CGI	Common Gateway Interface
FLOPS	Floating Point Operations per Second
HTTP	Hypertext Transfer Protocol
HTTPS	HTTP Secure
NAT	Network Address Translation
REST	Representational State Transfer
RPC	Remote Procedure Call
SCP	Secure Copy protocol
SSH	Secure Shell
URL	Uniform Resource Locator
VCS	Volunteer Control Server
VCS SD	Volunteer Control Server Scheduler Dispatcher
VCS WPM	Volunteer Control Server Worker Pool Maintainer
VW	Volunteer Worker
VW FM	Volunteer Worker File Manager
VW SM	Volunteer Worker State Manager
VWP	Volunteer Worker Pool
WS	Worker Surrogate
WS OBSWM	Worker Surrogate OBS Worker Manager
WS VWM	Worker Surrogate Volunteer Worker Manager
XML	Extensible Markup Language

Used Terms Explanation

Dispatcher	Component of OBS, assigns jobs to workers
M-Voting	Validation system consisting on distributing a job among a group and wait until “m” results are equal
Repository Server	Component of OBS, manages binaries and related tasks
Scheduler	Component of OBS, calculates the job queue
Source Service Server	Component of OBS, provides services related to source code changes
Spot-Checking	Validation system consisting on using a well-known job to detect malicious volunteers
Validator	Component of second approach design, manages results sent by volunteers
Verifier	Component of first approach design, manages results sent by volunteers
Volunteer Cluster	Group of Volunteer Workers performing the same job
Volunteer Worker	Component of second approach design
Worker	Component of OBS, a program that builds software packages

1. Introduction

The high cost of early computers versus the processing power needed, soon highlighted the need of new computing ways. Distributed computing on its first steps made possible the achievement of first supercomputing projects, introducing the concept of “cluster computing”. Here, the idea of using multiple relatively low-end computers to reduce costs to do a more complex task was born.

On the late 90's a different way of distributed computing, called "volunteer computing", started to be employed for projects. The main idea was analogue to the old one: different standard computers share the costs of a heavy and specific task working concurrently on it. That was possible because of two main factors: the access of the general public to personal computers, and the improvement and expansion of Internet connections.

Nowadays, we live in an era that we all have access to extremely powerful computers and every device is permanently connected. Therefore, volunteer computing might become the best way to dedicate our so often underused processors to collaborate on giant projects for the general interest.

As an example, we can talk a bit about BOINC (Berkeley Open Infrastructure for Network Computing) platform. The project started in 2002 as a base for the existent SETI's volunteer computing project, and today it groups more than 80 active projects and has almost 600.000 contributing computers.

One of the most demanding computational tasks is software package building and compiling. The work itself has been optimized during years by task automation. However, in recent years the distribution of the jobs has proved to be the best way of reducing costs.

On Linux based systems, the amount of users that can collaborate by spreading their own improvements and changes on the distributions makes compiling or building packages or distributions a very heavy task that the Open Source community usually cannot afford. For that purpose, systems like OBS (Open Build Service) have grown in parallel to repository systems to make this work easier.

OBS is a system where collaborators around the world submit their changes to repositories, and then a worker's host structure does the compile and build job for them in a distributed way. However, this OBS system is limited to divide the tasks in workers that reside in trusted hosts part of the network that runs the OBS instance.

As a consequence, the research applies the idea of volunteer computing to package building, particularly to OBS system. It provides new challenges based on the new pool of heterogeneous, untrusted workers: the reliability, scalability, performance and security of the system.

So, here appears the main contribution of this thesis: a design of a process to apply volunteer computing to OBS by modifying or attaching something to the actual system. A proof-of-concept is implemented also showing the basics of the new design.

The following chapters are structured on this way: Chapter 2 introduces the concept of volunteer computing and shows the current research on it, presenting the existing platforms and different approaches to it. Chapter 3 introduces OBS, software package building and describes the problems, solutions and processes to accomplish the task. Chapter 4 goes deeper on the chosen design specifying it. Chapter 5 explains the implementation work done on successive phases of the work. Chapter 6 makes the analysis of the results obtained and process developed. Finally, Chapter 7 draws the conclusions about the Thesis process and the solution itself.

2. Volunteer Computing

2.1. Concept

Volunteer computing is a form of distributed computing where the general public volunteers processing and storage resources to computing projects. [1]

Distributed computing is a way of computing where different components of a distributed system collaborate between themselves to achieve a common goal. In addition to volunteer computing, there are two main options to lower costs of big computing projects.

One of them, is Grid Computing, where different systems of different institutions are coordinated to carry out bigger projects. This is a way to share the computing power of different communities by investing little money to manage and maintain all the architectures in common.

Another way is utilizing cloud services for these purposes. In Cloud Computing, the users gain access by the network to a shared pool of configurable computing resources. In this case, the cost of deploying and maintaining the structures is saved up, but the communities still have to pay for the computing time used.

From this point of view, Volunteer Computing provides the advantages of cloud computing in terms of no physical structures needed in addition to free access to computing power. However, it puts forward some new challenges that we will discuss later. [1]

The real distributed computing concerns us dates back from the early 80's. However, in this fifteen recent years, volunteer computing has presented an important growth [2] because of three main factors.

The first one is the improvement on processing power available. Following the Ahmdal's Law, nowadays each single user has on his pocket more computing power than a supercomputer could perform twenty years ago.[3]

The second one is on part a consequence of the first one: if twenty years ago, computing was a scientific, student and early-adopter thing, nowadays almost every family of the first world owns one or more computers. The high spread of this technology makes that more users have more processing power available.

The third one is the evolution and spread of the Internet connections. From 56k-modems on dial-up connections around Europe fifteen years ago, nowadays we can talk about each family owning a minimum of 1MB-ADSL connections at home. So, the existence and speed of the data lines linking millions of users around the world, makes each one of them a potential volunteer.

However, managing this growing pool of potential volunteers involves many new challenges to address with. While other distributed systems depend only on technical users and developers prepared to treat with complex processes, volunteer computing relies on standard users to achieve its goals. Three main issues have to be taken into account: accessibility, applicability and reliability.[4]

Accessibility means trying to make the process of joining and collaborating on a project as easy as possible for the user. It involves, at first, ease-of-use and platform independence: each user has to be able to become a volunteer only following a few steps of installation. It should not matter what platform or architecture is he using. At second, the security for the volunteer is one of the things that have to be attended to not discourage users from joining. They have to be sure that the software they are going to execute will not be harmful for their computers. The third one concerns the user experience. This means that the user has to receive some kind of feedback from the project, not only a good user interface but a way of make him feel involved in the task.[4]

Applicability means that the volunteer computing system has to be useful and effective for the task it is performing. At first, the term “adaptive parallelism” can be used here, meaning that many different systems have to perform well working in parallel. Because of the evolution of architectures and its heterogeneity, the system has to be prepared to work on every case. It should not matter who joins or leaves the computation. This point is very related to performance and scalability, because we need our system to exploit at maximum the processing power that our volunteers offer to us. At last, and concerning the volunteer computing platforms that support different projects, the system has to offer programmability to allow the implementation of different applications to run inside it. [4]

Reliability means that because of its open nature, volunteer computing systems are more given to suffer faults than other distributed systems. [4] Not only the possibility of unintentional crashing or leaving of the volunteers, but also the submitting of erroneous results to the system, intentionally or not. So, the system has to be fault-tolerant and in some cases sabotage-tolerant, to prevent malicious users from corrupting the system. There are many different approaches to improve the reliability of a system that we will discuss later together with the existing research on the field.

Some of these issues have become technical requirements for the existing volunteer computer systems.

2.2. Current research and existing platforms

Now we are going to discuss the different technical requirements of a volunteer computing system. We can assume that nowadays the current model of the system is based on the *master-worker paradigm*. Here, the master divides the massive task into small pieces. These pieces are distributed to the different workers, that carry on the required computation and send the results back. Then is time for the master to verify the results and group them to compute the final product. [5]

To describe the different approaches to the requirements we are listing the different existing platforms for volunteer computing. Here, the term “middleware” appears, because it is the piece that fits in the middle between the volunteers and the master of the work. It allows the communication and distribution of resources between all the parts. Sometimes a middleware is just a platform where to deploy the different volunteer computing projects, and sometimes each project has its own middleware.

As an example of the present growth of volunteer computing, we are going to discuss the similarities and differences between five different middleware platforms. We will show how they solve the different technical issues of volunteer computing. These are: BOINC, XtremWeb, XGrid, GridMP and the project Folding@home.

2.2.1. BOINC

The Berkeley Open Infrastructure for Network Computing project was created in 2002 to develop open-source, general-purpose volunteer computing middleware. BOINC provides server software that lets scientists create volunteer computing projects, and client software, available for all major platforms, that lets volunteers participate in any combination of these projects. [6]

As a volunteer computing system, three main aspects have to be taken on account: the data model, the computing process, and the communication model.

The data model of BOINC is based on files. They can be input, output or components of applications. Each project stores the physical files which are linked by each running job to optimize storage. All the files are digitally signed using the private key of a public key given to the client, preventing the distribution and execution of malware. [7]

The computing model of BOINC is based on the following abstractions. Each submitted job, corresponds to an application, that is an algorithm that performs the task. Each application has different “App versions”, that correspond to the different algorithms executed by different platforms(for example, Windows/Intel32). In addition, each platform-application combination, can have more than one version, depending on the configuration of the processors executing the job. That is specified by a “plan class”. [7]

When BOINC scheduler is considering to send a job to a host, it uses a version selection algorithm to decide which app version use. [7] Each BOINC job includes information to help the scheduler to decide about dispatching.

As we have said before, volunteer computers are untrusted by definition. This means that BOINC needs to include different tools for ensuring that the final submitted results are correct. Some results have properties that permit a quick verification after received. In other cases, BOINC has to use replication to verify results, dispatching the same jobs to different hosts. However, some floating-point results can be slightly different when run on different processors. That is why BOINC implements fuzzy comparison for these cases. Going further, some results that could be numerical unstable, like physical simulations, need the so called Homogeneous Redundancy to verify the results.[7] The main idea, is to ensure that each job is executed on the same kind of processor using the same numerical libraries to obtain the same results.

The communication protocol of BOINC is based on the use of HTTP and HTTPS protocols, and the initiation of communications by the client. Each project is identified by the URL of the website. This page also contains the URLs of its schedulers. When the client is connected to the project, the URL is redownloaded periodically to maintain connections to the scheduler RPCs. This is BOINC's central protocol. [7]

As can be seen in Figure 1, a BOINC server consists of a set of Web services, a set of daemon programs that communicate through a relational database and a set of utility programs. The scheduler, acts as CGI and handles RPC dispatching jobs to the clients. It gets the jobs from a shared memory cache, maintained by a feeder program. It is also the responsible for different policies to avoid malfunctioning hosts and improving performance. On the other hand, we can find the different daemons; *work generator*, that controls the flow; *transitioner*, that centralizes database updates; *validator*, that compares the set of completed jobs; *assimilator*, that handles the validated jobs by storing them; file *deleter*; database purger that maintains the database dropping tables; *trickle-up message handler* from applications. [7] This server is designed for scalability. All the different components can be moved to different machines. All the daemons can be replicated, handling different groups of jobs.

The BOINC client consists of some components. These are the *Core Client*, the *Manager*, the *Screensaver Coordinator*, and the *default screensaver*. The Manager and Screensaver Coordinator are basically the GUI of the client. They communicate with the core client by RPC interfaces, using XML messages on a TCP connection. The main tasks of the core client are two: job scheduling and work fetching. This follows the calculations made on the job scheduling phase. [7]

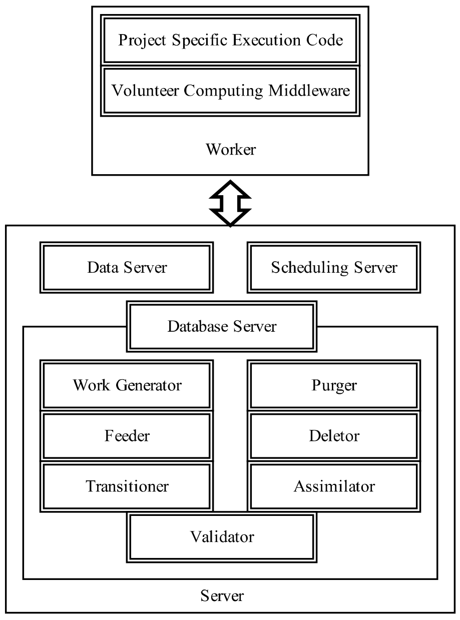


Figure 1: Client and Server Side of BOINC

2.2.2. XtremWebCH

XtremWebCH is a derivative of the generic global computing system called XtremWeb specifically designed for volunteer computing. It is an open-source middleware that allows to easily deploy and execute parallel and distributed applications on a public-resource computing infrastructure. [8]

As can be seen in Figure 2, the main architecture of XWCH concerns four modules: *coordinator*, *client*, *worker* and *warehouse*. [9] Also the communication model and protocol differ from the pure volunteer computing system, because it takes ideas from the *peer-to-peer* paradigm.

The coordinator module is the central element. It provides the web interface to manage and monitor the process so acts as GUI. It also maintains the lists of submitted jobs and connected workers.[9] With this information, it assigns the jobs to the volunteers, and keeps track of the process to ensure that is finished properly.

The client program is responsible for sending "computation requests" to the coordinator. [9] So, depending on its rights, it is the module capable of submitting jobs, and saying what files they will use.

The workers are the computing nodes running on the volunteer computers. Once the worker is running, its first step is to contact the coordinator for registration. It sends the information with its performance and characteristics. After that, the worker node is able to accept jobs, obtain files, do the computation and send back the results. [9]

The warehouse module is used to maintain the repositories of the needed data and executables.[9] Use to be some instances of them per deployment, easing results submitting by the workers.

The communication between the workers and the coordinator is always initiated by the first ones, to avoid firewalls and NAT. They use four types of signals: *WorkRegister, WorkAlive, WorkRequest* and *WorkResult.* [9]

The communication between clients and coordinator is supported by an API that allows job submission by a users' service. This allows that instead of having prefixed jobs attached to a project, once the client is registered in the system, it can generate applications and jobs dynamically.

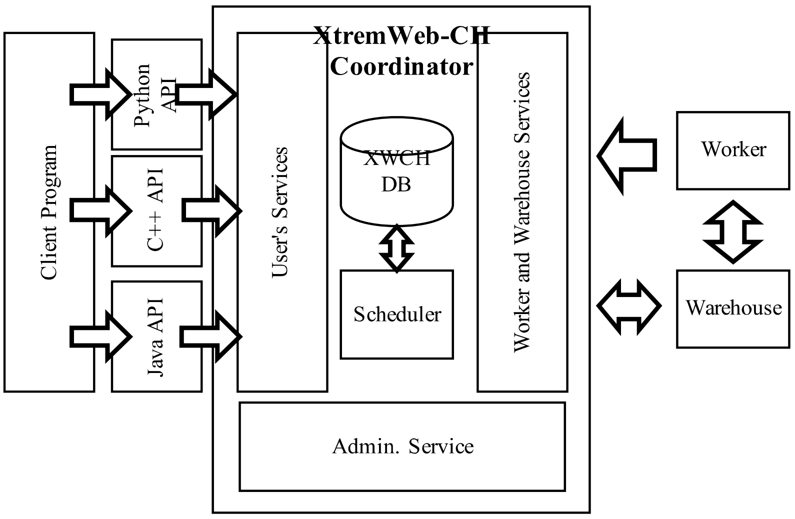


Figure 2: XtremWebCH Main Architecture

In addition, one of the features that has the influence of peer-to-peer networking is the possibility of data replication. When an output file is generated by a worker, he can choose to distribute many copies of it to different warehouses. This helps in case that the output file is needed as an input to other jobs. Also, the workers can get these results directly from other workers, easing the distribution of files.

The data model is also important for a volunteer computing system. The first abstraction is the *Application,* what is a set of *Jobs.* is the execution of a binary file on a given worker. The third one is the *Module*. That is a set of binary codes having the same source code, but targeted to different platforms(OS/CPU combination). [9]

Regarding that information, we can say that XtremWebCH is not as pure volunteer computing platform as BOINC, because it still has some lacks of accessibility and trustfulness. On the other hand, it takes some of the advantages of performance and efficiency of the pure grid computing systems.

2.2.3. Xgrid

In 2005, Apple released Xgrid middleware as a new technology solution for loosely coupled, distributed computation. It is based on Zilla.app, created by NeXT in the late 80's. As an Apple proprietary product, it has been promoted as an extremely usable distributed computing solution for less technical users. [10] Since 2012, concurring with the launch of OS X 10.8, the platform is no longer supported. However, it suits for our purposes of showing an easily deployable volunteer computing solution.

As can be seen in Figure 3, Xgrid has three level architecture with many similarities to XtremWeb. It consists of a *Client*, a *Controller* and an *Agent*, usually distributed.

The Client provides the user interface to the system, finds a suitable controller, submits jobs and retrieves the results. It has a very simple GUI supporting basic functions, like the information of Xgrid Agents and Controllers. The important part of the client is the Command Line Interface, that allows job submission and configuration. The support for developers to build applications for the system is made possible by the Xgrid Cocoa Framework, that allows the use of typical functions to handle the distributed computing. [10]

The Controller is again the main part of the Xgrid middleware, because it handles the communication between clients and agents. It interprets job submissions by the Client and decomposes them into small chunks following the specifications of the job about dependencies, payloads and deadlines. Then, it schedules and dispatches the jobs to the available agents monitoring the process and notifying availability. After, regarding the result of the task, it submits it to the clients or to the next agent.[10]

The last level on the architecture is the Agent. It is the one who actually executes the computational tasks of the jobs. Typically, one agent is running per machine(or CPU in some cases) and it is responsible to bind to the first available Controller of the network. [10]

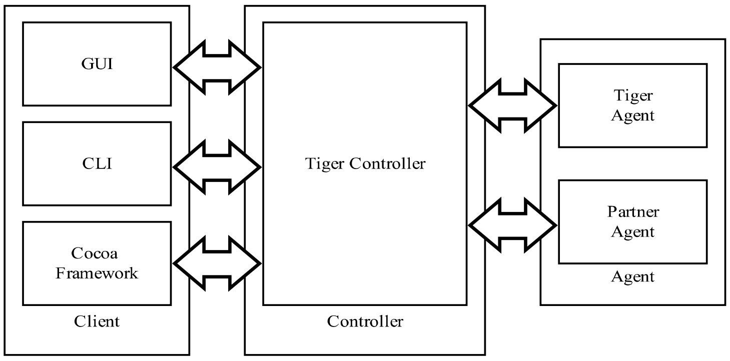


Figure 3: XGrid Native Architecture

The communication model plays a very important role in a distributed system. On it relies a huge part of the reliability and efficiency of the process. The protocol used by Xgrid is BEEP, Blocks Extensible Exchange Protocol, similar to HTTP but designed for full-duplex communications and peer-to-peer networking. It uses XML profiles to define channels over a socket, reducing the negotiation overhead. All communications between clients, controllers and agents are able to be encrypted ensuring security and privacy. [10]

We can say that Apple's Xgrid solution does not follow the paradigm of volunteer computing like BOINC does, however some of its principles are taken. It is more than just a grid computing system, regarding its proved accessibility and scalability.

2.2.4. GridMP

GridMP is a commercial distributed computing platform designed to improve the performance of computation-intensive applications. It also provides features for end-users, administrators and organizations that enhance the usability, reliability, security and scalability of the processes.[11] [12]

As can be seen in Figure 4, GridMP architecture consists on several components that collaborate to perform the computation. We could separate them in layers as workers, master and interfaces, but the description provided is somewhat more specific. The platform is composed by: *Database, Agent, Service Manager, Realm Service, Poll Service, Dispatch Service and File Service*. [12]

As an interface, GridMP has the MGSI(MP Grid Services Interface) that allows developers to use functions to perform the required tasks for the applications. This is based on XML and SOAP and groups both the File Service and the RPC service. Also, a *Management Console* is available to submit, monitor and manage Jobs. [12]

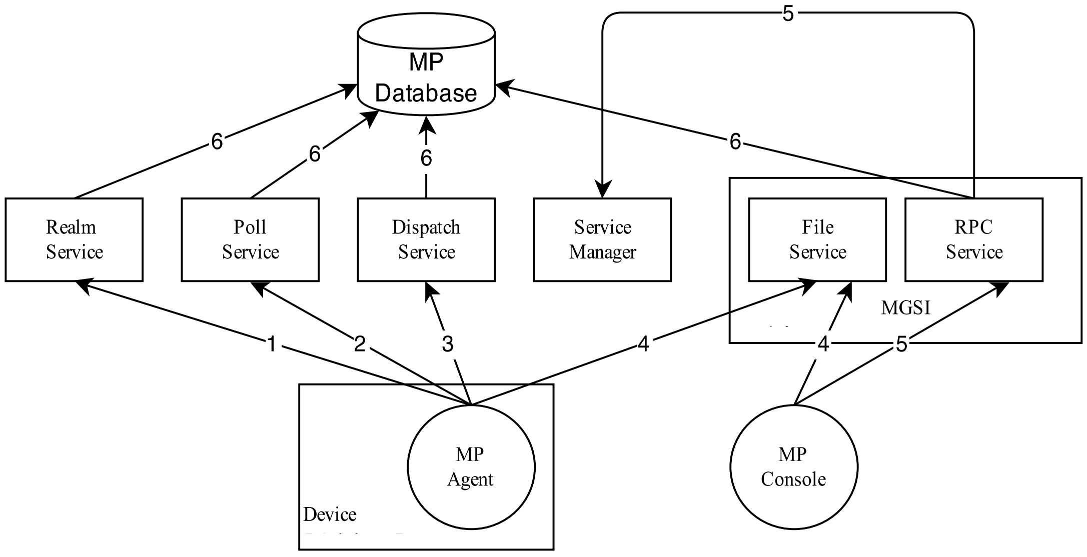


Figure 4: GridMP Components and Workflow

The flow of the processing between the cited components goes as follows. First, different devices connect to the Realm Service for authentication and receiving credentials. Second, these devices, now agents, contact the Dispatch Service to get assigned work. Now, the Agent requires and downloads the needed files from the File Service, and starts processing the work. The work is now being monitored as the Agent sends periodically reports to the Poll Service. When the work is finished, the Agent uploads the results to the File Service, and asks for more work to the Dispatch Service again. [12]

The work flow explained above needs to rely on a reliable and efficient communications model. This model is based on encrypted XML-RPC or SOAP over HTTP or HTTPS depending on the case. [12]

As the data model, we can talk about levels. First, the application and program objects hierarchy: An *Application* consists of a set of Programs. Each *Program* can take part on one or more applications. It consists on the executable code sent to the devices. These Programs allow multiple versions and modules, to support code modifications and different architectures and OS. [12]

The second level refers to *Job* and *Data* objects. Each Job is associated with an application, and it groups one or more Job Steps, that is the instance of the execution of a Program. On the other hand, Data files are grouped on *Data Sets* and can be used globally or associated to a *Job Step*. Now comes the smallest piece of work to schedule, called *Workunit*. Each one belongs to a single Job Step, taking Data files, and producing Results. [12]

As we have said before, GridMP is just another option to enable volunteer and grid computing for organizations. Although is somewhat more complex than Apple's Xgrid and much more focused to Grid computing than BOINC, it gives a middle-solution in terms of usability, reliability, security and scalability.

2.2.5. Folding@home

Folding@home is a distributed computing project that helps researches about protein folding, drug design and molecular dynamics. Although it was launched in 2000, it is the fastest volunteer computing system, grouping approximately 18 petaFLOPS of processing power.[13]

As can be seen in Figure 5, the main architecture of Folding@home project consists on four main components, and a Web server acting as the interface. These are the *Clients* and computational cores, the *Work Servers*, the *Assignment Servers*, and the *Collection Servers.*

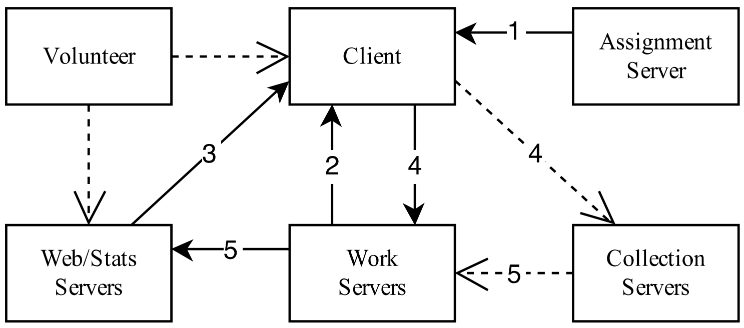


Figure 5: Folding@home main architecture and workflow

The Client has two main roles: first, it iterates between asking for work units, processing them, and returning the results; second, it interacts with the volunteer keeping him informed about what is happening on his computer. The software for the volunteer has another part, called core, that is responsible for the computation itself, and can be updated securely without the intervention of the volunteer. Cores also save the state of the computation preventing loses caused by the crash or the suspension of the works. This separation of the client at two levels allows the setup to be kept as simple and intuitive as possible for the volunteer. At the same time, the executable code can be very specific for the computational project.[3]

The Work Servers have the role of dispatch work units for the projects to the clients, and then accept, store, log and analyze the completed work. In projects of this kind, the planning of the working process is the most computational intensive task for the server, because on it depends the next improvement of the model, and has to take in account the existing ones. [3]

So, the task of acting as a global scheduler remains on the Assignment Servers. These are contacted by the clients and they decide regarding on deadlines, CPU power and volunteer preferences, the way to balance the workload.

The Collection servers, that allow a Client to upload results in case that a Worker Server is offline. [3]That improves the performance of the whole system, because first, the Client can continue working, and second, the Worker Server is not flooded with requests from the waiting clients.

As we said above, there is a Web part of Folding@home that acts also as the interface. That hosts the statistics and forum website. The system encourages volunteers to collaborate by a credit points system that accounts the participation of each volunteer.

It is not a coincidence that Folding@home is the Volunteer Computing project that groups more computing power. It provides usability and accessibility to the users and flexibility to the developers and researchers. By the nature of its computations, the results can be trusted by themselves. Also, the separation between Client and Encrypted core ensures security for the volunteers. Folding@home has an extremely flexible architecture that could support many other kinds of projects. This is opposite to BOINC that provides a standard client, server and statistics server, but with a fixed architecture that limits the types of projects that it can accommodate.

2.2.6. Conclusions: suitability for Volunteer Computing

All the shown platforms could be used to support Volunteer Computing projects. However, they differ in many aspects that affect directly their suitability for various purposes.

We can find the “pure Volunteer Computing” middleware looking at systems as BOINC or Folding@home. Both are prepared to work with completely untrusted workers. The security levels from the volunteers to the server and vice-versa are very high.

The protection of volunteer's computers when attached to the system and the reliability of the computation and the obtained results is ensured by default.

XtremWebCH, Xgrid and GridMP have very interesting features and implement good authentication systems, but they are not prepared to support an enormous number of untrusted volunteers and the systems may become unreliable.

Regarding the applicability and accessibility of the different platforms, possibly Xgrid from Apple is the one who cares most of ease-of-use and ease-of-deployment. BOINC, XtremWebCH and GridMP are reasonably prepared to be deployed on different circumstances. However, BOINC seems to need less configuration work to be applied to pure Volunteer Computing. Although Folding@home claims to be more flexible for the kind of projects that can support, it is not presented at start as a middleware, so we suppose that the process of adapting its architecture will not be as standarized as the others.

Finally, we need to make a distinction between commercial and non-commercial software. Both BOINC and XtremWebCH are open-source solutions that you could download and deploy following the instructions. However, this means also that ease-of-deployment should not be expected. Folding@home is a system that belongs to Stanford University, but is running over non-commercial licenses. Xgrid and GridMP are commercial software but are focused on different ways. When Xgrid is software usually included with Mac OS packages and using its licenses, GridMP is a complete Distributed Computing solution that includes many different licenses and support.

3. Applying Volunteer Computing to OBS

3.1. Open Build Service

Open Build Service (previously known as OpenSUSE Build Service)[14], or OBS, is an open-source, cross-distribution development platform published under the GPL [15, 16] .

It is used to build packages for different Linux distributions running over different target architectures. An OBS instance can be deployed by any person or entity although there are many of them currently available.

Usually, OBS is used to configure, build and publish packages by the developers. It includes automatic dependency resolving and linking to other projects. Each change in a depended package will trigger a rebuild on the depending package [14].

The OBS services have a distributed structure, composed by front-end, back-end, storage node and one or more attached worker hosts.

The front-end consists on the API for interfacing applications. It is accessed by web-based applications, command-line interface or XML-formatted messages [14].

The back-end hosts the repositories of packages, manages the scheduling of the building process and maintains information about the build hosts. There are many components inside the back-end, like Scheduler, Dispatcher, Publisher and Repository Server [14].

The Worker Pool, usually hosted on one or more machines, contains the build clients that actually perform the computations. Each build host can run one or more workers at the same time and build for one concrete architecture [14].

The Storage Node is an external part of the system that maintains the source code repositories, notifying the back-end when changes are made [14].

3.2. Software Package Building

A software package is a piece of compiled source code that can be run on a specific operating system using a packet manager system. This source code is usually built for different hardware architectures and distributions.

Packages can depend on other packages to perform their activities. These dependencies can be both build-time and run-time dependencies. Here, we are going to concentrate on the first ones.

Package building for repositories is a heavy time-consuming task. The tree-style organization of dependencies has its advantages, but also provokes that when a rebuild is triggered, the number of packages to rebuild grows on each new branch reached.

Also, the task cannot be completely parallelized, because some packages need another ones to be built before them. This situation may sometimes cause bottlenecks, when small packages need to wait until the building of a big one is finished.

As a result, the package building can take from minutes for the lightest ones, to days for the heaviest ones. This affects the pace of development of new software, because sometimes for a small patch modification, a big system needs to be rebuilt to test it.

The Open Build Service initiative at least centralizes the software development and help developers to manage packages and dependencies at the same time [16].

3.3. Thesis purposes and issues

By its nature, Open Build Service is a limited system that provides open software developers with a service to build and package its work among distributions. At this moment, all the costs of supporting this platform are assumed by the instance owner. This means processing power of the build hosts, storage and bandwidth.

However, if we believe that OBS is a necessary tool for the spread of open software, the scalability of the system is a major issue. On a non-commercial community, the resources to support this are often limited. That is why the effort could be distributed among uninterested contributors.

The solution here is simple: OBS could benefit from allowing anonymous users to work as build hosts. That is, to apply the concept of Volunteer Computing to Open Build Service. As Folding@home uses the processing power of thousands of volunteers to support research computations, OBS could do the same to build packages for the open software community.

The most important problem appears when talking about the reliability of the new system. As new build hosts may not belong to our organization, there is no guarantee that the performed work is correct. So, the new system should be able to address with problems of this kind.

After, we could list what we call applicability issues. The existing volunteer computing systems can support many kinds of projects. Some of them require a huge amount of processing power to be carried out, others need an important exploitation of the bandwidth, and a few more also require the use of the volunteer's storage capacity. However, building and distributing software packages is an overall intensive task that needs CPU and bandwidth consumption and sometimes storage space.

The last kind of problem is the one related to the accessibility of the system. As we are not contracting them, the new OBS system will require ease-of-use and trust from the volunteers to the system to be as spread as possible.

Regarding both that list of issues and the different existing platforms we described above, the decision of developing the new system has to be done. We know that none of XtremWebCH, Xgrid or GridMP should be selected because of their lack of fault-tolerance. Folding@home also has the problems of not being a “pure” middleware: when some ideas can be taken from its architecture, the system itself is not prepared to accept OBS.

Only BOINC remains, that is the most universal, open, volunteer computing system middleware that exists. However, the kind of projects that it uses to support are very different from what we need to OBS. Also, the architecture proposed by BOINC goes more towards an integration of the OBS structure into BOINC rather than being only an addition to the current system. When some Volunteer Computing projects migrated to BOINC architecture because of its similarities and advantages, we think that it is not worth for OBS to do that.

3.4. First Approach: OBS Architecture Modification

The first approach when an adaptation from one system to a different concept is needed, consists on modifying the original one to fulfill the new requirements. Following this path, we need to perform several modifications on OBS system to adapt it to the Volunteer Computing concept.

3.4.1. Studying the original system

The target system needs to be known, identifying its different components and understanding their roles on the work flow.

The worker, once it has been initialized and its connections have been configured, requests the worker code from the repository server to keep it updated. Once restarted, the worker starts sending state messages every five minutes to the repository server. The repository server maintains a list of all workers available on the worker pool.

As we have said before, OBS triggers a rebuild each time a user updates the sources of a package. So, when the *Source Service Server* notifies a change, the *Scheduler* will schedule the build jobs in the job queue. There, the *Dispatcher* will find the new jobs and assign them to the workers through RPC calls.

Each worker will download the needed files from the *Repository Server* and start performing the job. After completing it, the worker will send back the result files to the Repository Server. The Scheduler now, will notice this change, so it will recalculate the dependencies and send events to the *Publisher* about it.

The Publisher, when receives an event from the Scheduler, will take the package, generate the related metadata and upload it to the download servers so it is available for the users.

3.4.2. Component Modification

Applying the concept of Volunteer Computing to OBS does not require only modifying the components of the back-end. It also requires implementing a complete front-end system that follows its principles and it also allows the collaboration without disturbance with the old OBS instances. However, we are going to focus here only on the main components and work flow of the back-end architecture.

An addition needs to be made to the current system. As the worker will reside on the volunteer host, we need an entity to act as “proxy” between it and the Repository Server. That will be the *Verifier*, and it is responsible for receiving the different results from the workers, perform the validation process, and send the chosen ones to the Repository Server. Also, it needs to communicate to the Dispatcher to obtain information about volunteer workers.

As a consequence, some other entities involved on the process need to be modified too. The *Worker* will be prepared to perform tasks as usual, but instead of sending its job results directly to the Repository, it will do it first with the Verifier.

The Repository Server needs several changes, as it is one of the core components of the system. It still should allow workers to make requests, but in order to keep the system safe, all the incoming data from them should be blocked. These communications need now to be done with the Verifier. Also, it needs to maintain more information from the Workers trying to ensure reliability, like authentication data and trust values.

The Dispatcher also needs to be modified because to allow validation, the jobs of the job queue cannot be attached on the same way to the Workers. At first, it needs to take information from the Worker Pool in the Repository Server, so it can decide which workers select to perform the job. After, to allow for example, validation by replication, it should distribute the same jobs over different groups of workers and inform the Verifier about it.

On Figure 6, the main components of the back-end architecture are represented, and needed changes can be seen highlighted too.

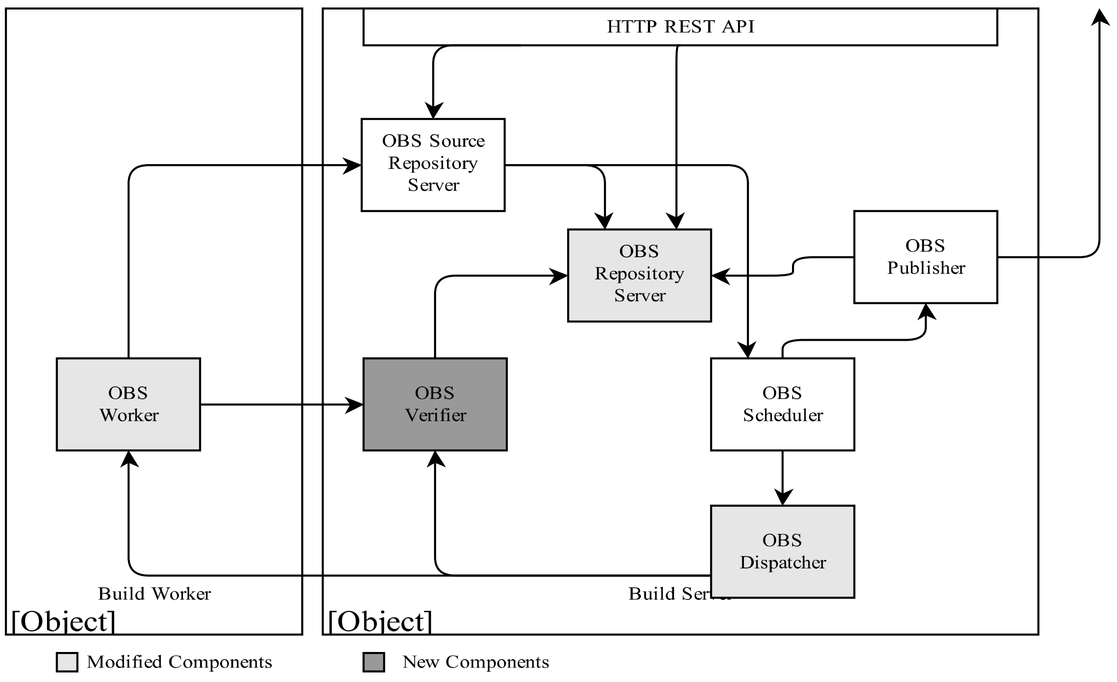


Figure 6: OBS Back-End Modifications

3.4.3. Known problems of the design

Regarding the last figure, it may seem easy to perform a few changes on the components and make it run with the new characteristics. However, dipping into the actual OBS code, the development aspect of the design starts to seem very different.

At first, OBS is a collaborative free-source code composed by many branches. This makes it an old code with modifications from different points of view that might be difficult to modify for our purposes.

At second, keeping our add-ons maintained at the same time as the original code grows is not an easy task. It may cause problems because of the number of collaborators working at different paces.

Also, the coexistence of both branches/projects on OBS community is a big issue to take care about. Some organizations might not want to use the new additions and others might see this as the only way to use OBS. However, the most part of the acceptance comes from those who will desire a flexible system combining both of them. As said before, maintaining a new system at the same pace as an old one is being developed by the community, could bring along a great number of problems.

Finally, as we showed above, Volunteer Computing is not only a concept, but a whole philosophy with the points of view faraway from the OBS ones. This means, for example, that not only the back-end will need to be modified. Also the front-end, including website access and community will need changes to accept and attract the new non-technical volunteers.

3.5. Second Approach: OBS Add-on System

Discarding the idea of modify an open-source code that has been and keeps changing at fast pace, a new approach to the design appears. The approach is to take advantage of the interconnection possibilities that OBS offers. Instead of a modified version of OBS, we offer them an add-on to support Volunteer Computing. This means designing an external system that offers trusted workers to OBS from volunteer workers; that is, thinking on OBS as a black-box.

3.5.1. Architecture and process in brief

Starting at this point, we need an entity to act as "proxy" between the communications coming from OBS worker, filtering that info and redistributing it inside our system. We call this the "*Worker Surrogate*", and in fact, it represents our "trusted worker" for OBS.

Also, our system needs a coordinator, that could communicate with the OBS instance, exchange information, and offer it a number of workers. It is called, the "*Volunteer Control Server*".

On a volunteer's computer, we also need another entity that actually performs the work, and receives and sends the files. This is called the "*Volunteer Worker*", and form groups called "*Volunteer Clusters*" that work as if they were only one OBS worker communicating with the Worker Surrogate.

At this point, we need also to highlight another entity, that we call "*Validator*", and is responsible for deciding which work is trusted to send it to OBS.

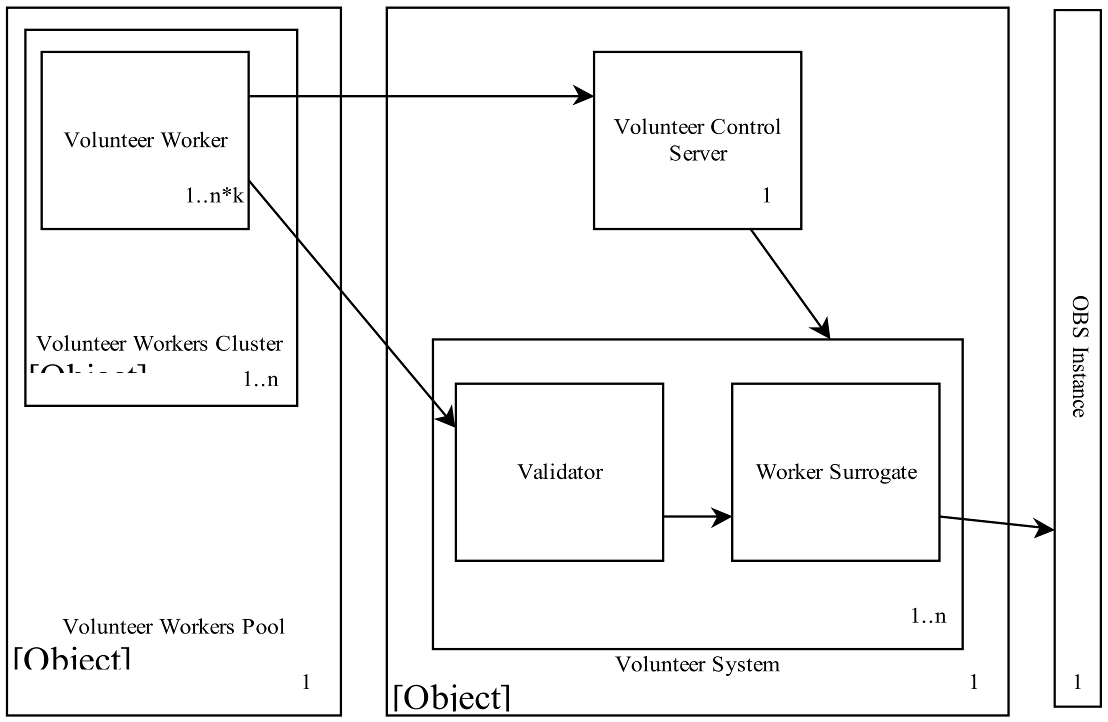


Figure 7: Overall Architecture

As can be seen in Figure 7, that system works as follows. The Volunteer Control Server is connected to the OBS system, and regarding some factors, it decides to offer a number of workers to perform the jobs. So, the VCS makes instances of Worker Surrogates to connect to OBS, and assigns each of them a number of Volunteer Workers form the *Volunteer Worker Pool*, grouping them as Volunteer Clusters. Also, it instantiates a Validator associated to each Volunteer Surrogate.

On the other part of the system, OBS notifies that a new worker is connected, so it sends a job to perform, that includes sources and binaries after some interchanges. Now, in our system, the Volunteer Workers have been connected to their correspondent Worker Surrogate and they are able to obtain the sources and binaries for computing. When the Volunteer Workers send the different results to the Validator, then is it which decides what is going to be the “trusted” work that we are finally sending to OBS. So, once the Worker Surrogate receives the files, it only has to transmit them to OBS being sure that the results are correct.

3.5.2. Considerations about the validation process

Like in all pure Volunteer Computing systems, the capability to ensure that the work we are obtaining as the result is trustful is one of the most important subjects. Some systems rely on the nature of the computational problems in order to do that. However, this is not the case of OBS, that needs to perform replication methods in order to accomplish that goal.

Here comes the validation process, that involves mainly the Validator and the Volunteer Control Server. Many options have been discussed before to decide which process apply, and we are going to explain them in this chapter.

The first part of it is to send the different results to the Validator once computed. Two options are, either send the files from the volunteers to the Validator or send hash lists to it. The main advantage of hashing are the savings on storage and bandwidth for the system, specially when we need a scalable system that allows massive replication.

Some issues appear here. The most important one is to keep track of the real file only obtaining the hash. This could have the disadvantages of either keep the volunteer idle until the process finishes, or make it store the files inside its computer somehow. The solution is that our process can have the advantages of hashing and at the same time do not lose efficiency or abuse volunteer's computers. However, there are some more factors involved on it.

The immediately related issue is if we are making volunteers come back to work just after sending its results, or we are going to keep them connected until the computation is solved. Here the answer is simple. Keeping our workers connected until the trusted result is ready has the advantages of better adjusting the trust values of each volunteer. However, it has the problem of being terribly slow because we are making the whole Volunteer Worker Cluster work as the slowest of the volunteers. As we have said, keeping track of the results later is not a problem.

The next question here is very similar but still has its different points of view. Once a Worker Cluster is formed, we should maintain it from one computation to another, or disjoint it and let the volunteers rotate as the system needs them. From an OBS point of view, the fewer times we connect, disconnect and change the surrogates the better, because we avoid spending time here and redistributing files, and we help its scheduler to make plans for the jobs to dispatch. However, talking about our volunteer system and the goal of ensure reliable and trusted results, the clusters are better being re-done after each computation. On that way we can balance the load and the trust levels.

Talking about the need for making Volunteer Workers wait for the Validator to decide the *m-voting* system needs to be explained. Basically, the Validator waits until a number, called “m”, of results coincide to send it to the surrogate as trusted by majority. However, there are some approaches to that, that depend on the fact of waiting or not for all results to send the definitive, and also to the size of m in comparison to the size of the group.

If m represents a number higher than the half size of the Volunteer Worker Cluster, waiting for all the results to arrive simply is not needed. The trusted computation can be sent when m-coincident results are reached, because more arriving results will not change the final one. However, if m is set as lower than the half of the group, a problem can appear: m-coincident results could arrive faster and, being wrong, to be sent to OBS as the trusted result. That is a very unlikely scenario that relies on not simply unreliable or malicious volunteers, but saboteurs working in collusion. We want here to support that if we take in account some facts as the listed above, the whole verification system has to slow down a bit to ensure reliability.

Notice finally that by waiting for results to arrive, we are only expecting to hold the sending of the last result until we could be sure of it, not keeping all the volunteers idle until it is done. That is something we are going to explain at the same time as the whole validation process.

One of the scenarios that could really slow down the validation process is when, specially working with small groups, the voting does not return any conclusion, and we need to request more workers to add results. Instead of waiting for the last hash of the volunteers to be received, the Validator could compare the number of results left to be received and the number of results still needed to reach m. On that way, it can anticipate the request of more workers.

It could also appear the case where there are not more workers available to join the computation. As the system is constantly rotating volunteers from one job to another avoiding them wait for the results, we consider that this might only happen on a general breakdown of the system.

Another very unfortunate scenario could be the crash of the worker selected to send the file, specially when is the last one. As we consider this very unlikely, instead of thinking on requesting files each time we receive a relevant hash, we prefer to, using our hash results, wait for the remaining volunteers or even requesting more if needed. Specially working with big files, the request of a transmission just to have a backup is something that needs to be carefully valuated.

One of the add-ons for the process that we want to include is the possibility to support blacklisting. Here, the Validator should receive when created by the Volunteer Control Server, a list of the associated workers that are blacklisted. In this case, when the hash of a blacklisted volunteer is received, the Validator will not allow it to count on the voting.

3.5.3. Considerations about sources and dependencies replication

On the process described above, the files needed to do the computations were distributed among the volunteers somehow. That part is now the object of discussion, because what to offer to OBS community with the system is one of the main points to clarify.

We can find three cases, each one with its advantages and disadvantages. Basically, the discussion concerns who is going to support the bandwidth load, and what counterparts has each choice.

The first case is what the majority of the people would expect regarding the brief process description. Here, the Worker Surrogate receives the binaries and sources needed for computation, and at the time the attached volunteers are connected, it is responsible for distributing the files among them. As we said, this is the way that fits better with the expected system. OBS intervention is limited to send the file to a worker as it usually does, so we are following the pure “black-box” concept. Also, this system is easy to be implemented to be reliable, because the distribution of the files relies on our own. However, this design has the counterpart that all the support has to come from our system. That is, the bandwidth overload generated by the replication is going to be suffered by the servers hosting the surrogates. That could be a major problem when talking about scalability of the system.

The second case is a variant of the first one. Here, the files are still being replicated one by one to the different volunteers. However, OBS is taking that load directly sending the sources and binaries. Reliability is still ensured, as the files come from a trusted system. Implementation is not more difficult than in the first case, because OBS allows direct access to sources and binaries from any machine. What our system should do here is just receive the information about the packages, and then redistribute it to the volunteers to make the requests. The most important advantage of this design is that the bandwidth load caused by replication is going to rely directly on OBS. However, we are not sure that this solution is always the best, because the OBS instances are usually deployed by open-source communities that may not be able to offer that extra bandwidth.

The third case could be, if correctly developed, the best solution to apply volunteer computing to OBS. It offers both computation and extra bandwidth to support it. However, it relies on new levels of trust checking and a complex system by itself that is out of our scope. The basic idea is, as in the first case, the Worker Surrogate receives the needed files, and then it redistributes them to one or more volunteers. However, the most part of the replication load is supported by the volunteers, because the files are being redistributed from one to another by Peer-to-Peer networking. That concerns many other challenges, not only deploying the system itself, but ensuring the reliability of the system by checking files from the surrogate to the volunteers and from one volunteers to others. Think that only one malicious copy distributed along the system may suppose the trusting of wrong results by the Validator.

The third case is the ideal design but as we said is far away from our scope. Choosing between the first and the second has many points of view. From a technical side, only the scalability of the number two could make us decide. However, if we really think what could be better for the system to have acceptance, there are many cases of use that may change our focus. Thinking about it as a service, where we provide trusted workers for an OBS instance using volunteer computing, forces us to choose the first design. On the other hand, if this system is just an add-on that enables OBS to use volunteer workers, we should not worry about bandwidth overloads and choose the second design.

3.5.4. Considerations about the Worker Cluster size

Another topic to discuss when going deeper in the proposed design is how to select the Worker Cluster size.

When the Volunteer Control Server is connected to an instance of OBS, it has to decide how many trusted workers offer to OBS regarding the Worker Pool and maybe other values. That is, how many groups of how many volunteers make do work on the computations.

The size of each group and the number of "m" is what concerns us now. If we assume that the reliability of the results is assured by the selection of m(in this project that is compulsory), we can say that the only difference between a small group and a big one is the speed of computation. Bigger groups are more likely to include faster machines, that could perform the tasks quicker. However, statistically and assuring a value of trust, bigger groups would have to wait for slower machines as well, so the general improvement may not be as high as expected. The only case that could effectively improve the speed of the process, is to know that a bigger group will not probably need to add more workers to the same task after collecting the results. This happens when m is not reached by any group, so we will not see any job redistributing its files again and wasting lots of time.

The second thing to take into account here is the information needed by the Volunteer Control Server to decide between big or small groups. It effectively knows the number of volunteers idle in the system, but once a minimum size of the groups is reached, it has to decide between more groups or bigger groups. This relies on some information about the OBS queue that has to be obtained. Regarding how many packages are pending to be built, and for what architecture, the Volunteer Control Server could decide the most efficient group size for each situation. However, as this "black-box" approach was chosen, we decide to avoid the rescheduling of the OBS tasks among our volunteers. This means, that we could balance our Worker Pool and the OBS workload on a general way, but never guess about package dependencies, priority tasks, and so on, as this is job of the original OBS scheduler.

Now we present the two options for managing the size of the Worker Clusters. The basic and less complex way is to determine before instantiation a number for the group size, that could assure reliability and is relatively fast. However, making that number change between each assignment of volunteers to OBS, could have its positive reflections on the performance of the system. These are both, the "Fixed" and "Flexible" approaches for the question.

From the point of view of the implementation, the fixed group size is easier, because it does not need the interaction of OBS. However, the flexible approach maintains the concepts of black-boxing, using only some statistics from OBS Also the implementation could not change too much regarding that on our process we already need the dynamic addition of volunteers to a task.

3.5.5. Considerations about trust adjusting

One of the improvements that theoretically could improve the reliability of a Volunteer Computing System, is the effective managing, adjusting and applying of different trust values for the different volunteers. However, our main considerations when developing the process make us not think about high-scale organized attacks to the system. Only to provide a reliable, fault-tolerant system that could also avoid malicious volunteers and bad workers collusion.

There are many ways to solve the problem of trust adjusting. Some of them combine not only the penalty for bad volunteers, but also the updating of values that show the efficiency of a worker when performing different tasks. However, our m-voting based system, only allows us to focus on keeping the unreliable volunteers away.

Moreover, to improve the efficiency and scalability of the systems, we are assuming some constraints on the validation process that make the perfect adjusting of the different levels impossible. We believe that it is better to gradually incorporate trust values to the process and make it work well, than treat to maintain a real-time trust adjusting and cause bottlenecks on the system. As on every volunteer computing system, a worker is not identified with a person. So, a malicious volunteer could notice his bad reputation and just change his identity without problems.

Considering this, the Validator may send the reports of unreliable workers once it notices that the hash does not match. As we stated on the validation process, the Volunteer Control Server may not notice and adjust that after some time. Also, because of the nature of our replicating system, some good and bad workers will not send, even finish, their results, what does not really help to keep track of everyone.

The entity responsible for keeping the trust values is the only one that stays connected to OBS all the time. This is the Volunteer Control Server, and once the Validator decides about results, he will receive the information of the different workers involved. So, when the next cluster of volunteers is formed, he can send the just-updated info to the new Validator.

Many systems could be considered when assigning jobs to volunteers based on trust levels, to obtain the most balanced worker cluster for each task. However, our system only keeps track of these workers that sent relevant wrong work to the Validator. On that way, the purpose is only to keep away from the voting system these workers that are clearly unreliable or malicious. One simple approach could be the “blacklisting”. This is an easy-to-implement system where the workers who reach a certain value of incorrect results submitted, are noticed as that, and put on a list that is shared with the Validators to act in consequence.

3.5.6. Considerations about build information

On every computer system, a minimum level of logging what is happening on it is compulsory. It helps developers and maintainers to measure the performance and understand certain behaviors. Specially when your system is providing a service or being a vital part of a bigger one, some information has to be kept from one execution to another.

So, our system, as an active member building packets for OBS, is required to log, at least, which packets have been built by what machines. Regarding that information, a maintainer could in case wrong results have been sent to OBS, take the appropriate actions to fix it.

4. OBS Add-on System Specification

Once all the considerations have been taken into account and the choice between the two main approaches has been done, we are going to explain in this chapter the design selected for the problem at this stage of development.

4.1. Main architecture

Following the architecture and the general process outlined above, a static view of the whole system needs to be given as a dataflow diagram.

As seen in Figure 22 in Appendix 1, three main groups of entities can be described. Each group is instantiated a different number of times on a real environment. Basically, there is one Volunteer Control Server per OBS Instance, that offers “n” Worker Surrogates having “k” Volunteer Workers connected to each them. This makes a total of “n*k” Volunteer Workers on the Worker Pool.

The Volunteer Control Server now is divided on *Volunteer Worker Pool Maintainer* sharing a memory structure with the *Scheduler Dispatcher*. The first one receives all the data from the new Volunteer Workers and keeps it updated for the Scheduler Dispatcher, responsible for launching new Worker Surrogates.

Each new Worker Surrogate is divided into three entities. At first, the *Volunteer Worker Manager* maintains the list and the connections to the Volunteer Workers passing them the different messages and requests. On the other side, the *OBS Worker Manager* can be found as the actual substitute of an OBS Worker, following the normal protocol used by OBS Server and workers. Finally, the Validator is now included communicating its requests and data structures to the other entities. It is also responsible for keeping up-to-date all the information concerning the hashing of results and their respective workers.

The last group is the Volunteer Worker and it is the one running on the different hosts used for volunteering. The *State Manager* is the entity responsible for following the protocol used by VCS to schedule and dispatch jobs and the *File Manager* is the entity prepared to perform the actual work and communicate with the Worker Surrogate.

Regarding the different data structures passed as messages between the entities on the chart, almost all of them appear explained there. The main memory structure, named as Volunteer Worker Pool DB, holds all the information concerning volunteers, surrogates and jobs. It is centralized on the VCS side, and the partial views needed by the different Volunteer Worker Managers are taken from them. On this way, data coherence problems between entities are avoided, and the processes only have to take care of maintaining it updated between the volunteer's host and the main system at an adequate pace.

4.2. State Diagrams of Entities

4.2.1. Volunteer Control Server

The Volunteer Control Server entities have been designed as standalone processes that perform different actions depending on their internal states.

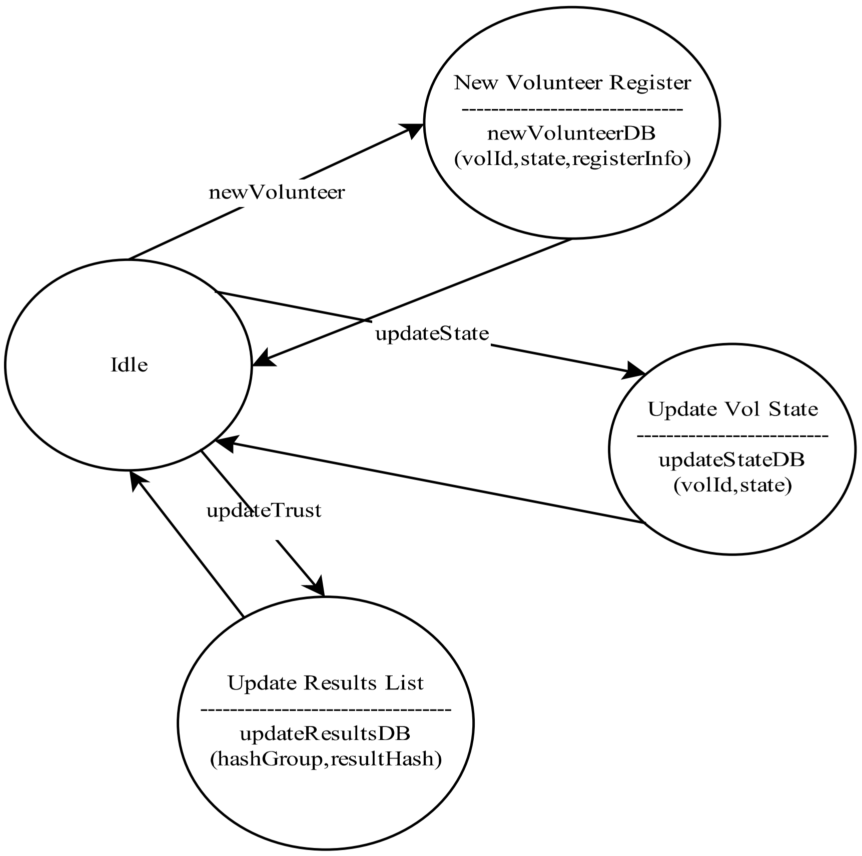


Figure 8: VCS Worker Pool Maintainer State Diagram

Figure 8 shows how the Worker Pool Maintainer keeps running idle listening to the different communications coming from the Volunteer Workers or the Validator. When a new volunteer is registered, or an existent volunteer changes its state, the WPM switches to a process to update the Worker Pool database in consequence. Also, when a new group of hash results is received by the WPM, it updates these database tables regarding the new information.

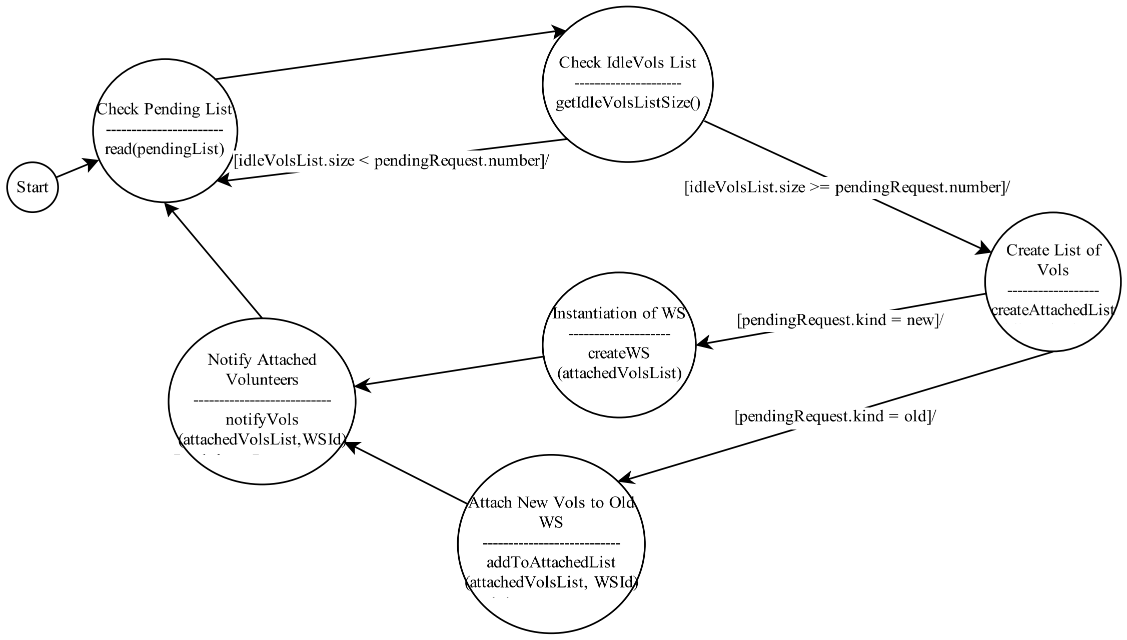

Start
Check Pending List
read(pendingList)
Check IdleVols List
getIdleVolsListSize()
[idleVolsList.size < pendingRequest.number]/
[idleVolsList.size >= pendingRequest.number]/
Create List of Vols
createAttachedList
[pendingRequest.kind = new]/
Instantiation of WS
createWS (attachedVolsList)
Notify Attached Volunteers
notifyVols (attachedVolsList, WSId)
[pendingRequest.kind = old]/
Attach New Vols to Old WS
addToAttachedList (attachedVolsList, WSId)

Figure 9: VCS Scheduler Dispatcher State Machine

Figure 9 shows how the Scheduler Dispatcher continuously looks at two different lists. The first one, called "*Waiting Job List*", contains the possible "request for more workers" coming from the running validators when they realize that the "m-similar" results cannot be reached. If not, that list contains the new group size calculated regarding OBS queue values. So, as expected, a flexible-group-size is used here, because new groups to dispatch jobs are calculated at each moment.

The second one is the "*Idle Worker List*". Once decided, the Scheduler Dispatcher creates or adds workers to an existing group, and comes back to start.

4.2.2. Worker Surrogate

Different to the VCS processes, all the Worker Surrogate entities work on a pipe-line-like way, because these processes are launched and stopped for certain purposes at certain moments. This short lifeline of the surrogates was decided when considering the rotation of the different volunteers among clusters.

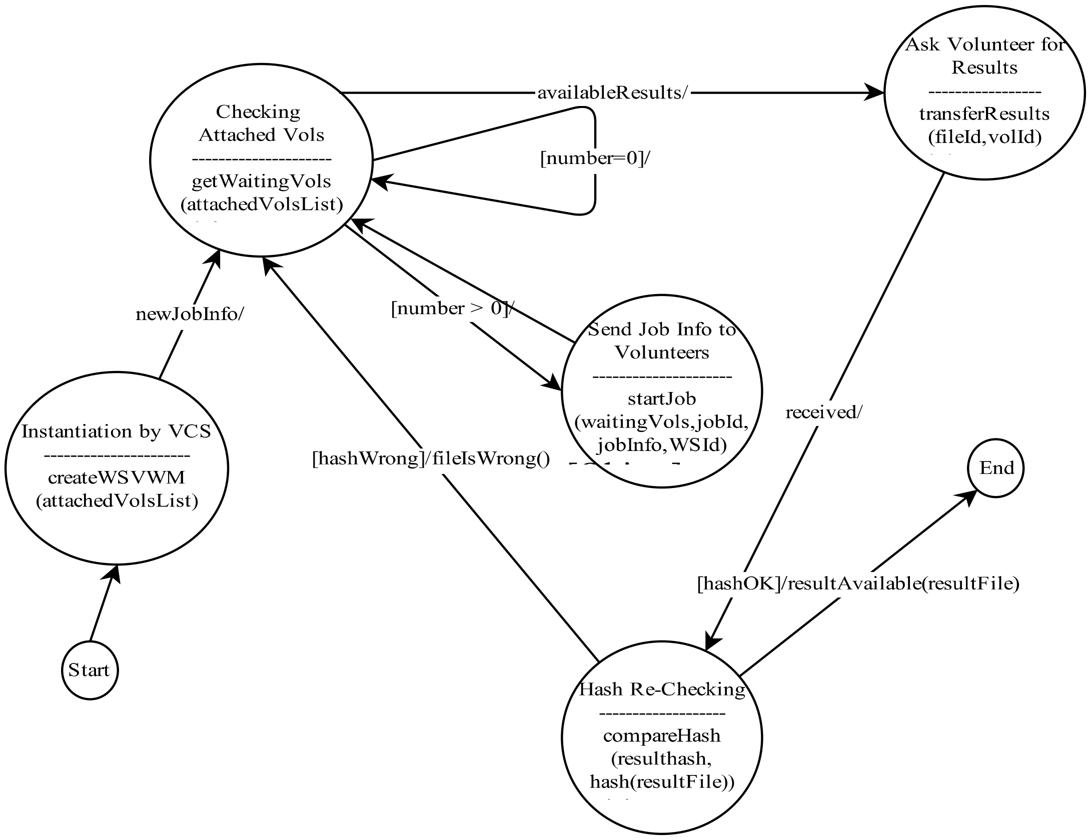

Checking Attached Vols
----
getWaitingVols (attachedVolsList)
availableResults/
Ask Volunteer for Results
----
transferResults (fileId,volId)
[number=0]/
newJobInfo/
[number > 0]/
Send Job Info to Volunteers
----
startJob (waitingVols,jobId, jobInfo,WSId)
received/
Instantiation by VCS
----
createWSVWM (attachedVolsList)
[hashWrong]/fileIsWrong()
End
[hashOK]/resultAvailable(resultFile)
Start
Hash Re-Checking
----
compareHash (resulthash, hash(resultFile))

Figure 10: WS Volunteer Worker Manager State Diagram

As can be seen on Figure 10, once the first volunteers are attached and the job information received, the process stays in a loop waiting for an available result notification while ensuring that all the volunteers receive the correspondent working requests.

Once the Validator notifies about a trusted result, one more check lasts: the Volunteer Worker Manager requests the file result to the volunteer, and compares both hash results again before sending it to OBS.

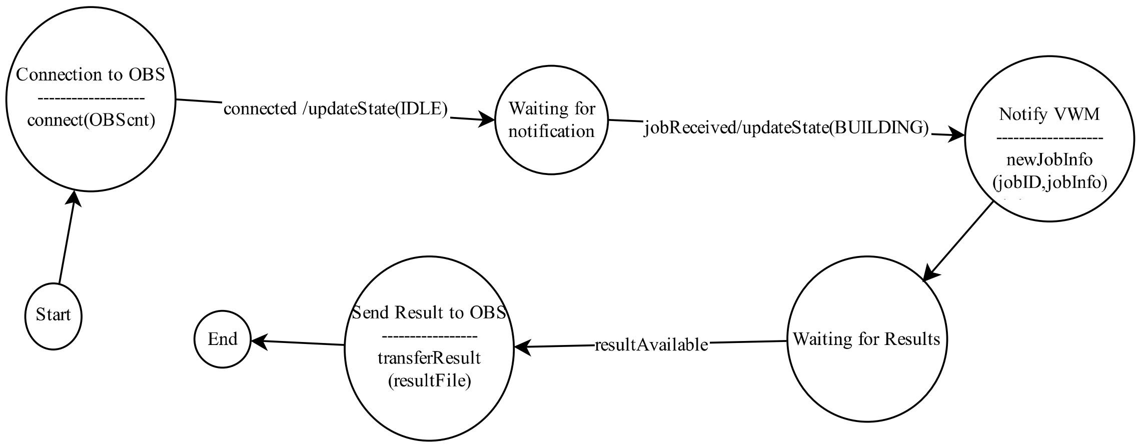


Figure 11: WS OBS Worker Manager State Diagram

As can be seen in Figure 11, the workflow of the OBS Worker Manager process is much simpler than the others. Its only responsibility is to send OBS Server state messages and job requests while the other part of the system performs the real computation.

One thing that was left to be decided on the considerations of the process was the issue of resource replication among volunteers. As can be seen on the figures, only the information needed to perform a job is requested, so the Volunteer Workers are the responsible entities of requesting OBS the actual files.

4.2.3. Validator

The Validator is an actual part of the Worker Surrogate. However, it was conceived as a separate one and its process needs much more explanation than any other.

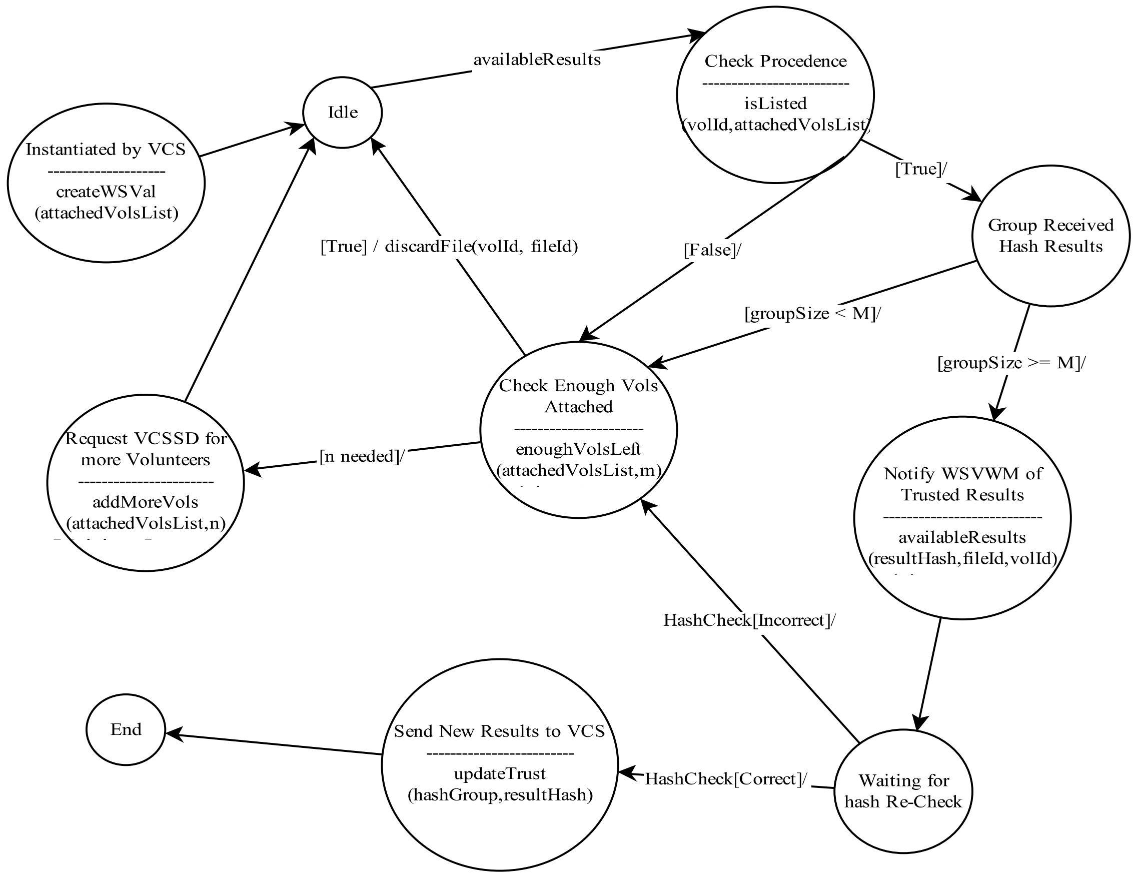

Instantiated by VCS
createWSVal (attachedVolsList)
Idle
availableResults
Check Procedence
isListed (volId,attachedVolsList)
[True]/
Group Received Hash Results
[False]/
[True] / discardFile(volId, fileId)
[groupSize < M]/
[groupSize >= M]/
Check Enough Vols Attached
enoughVolsLeft (attachedVolsList,m)
Request VCSSD for more Volunteers
addMoreVols (attachedVolsList,n)
[n needed]/
Notify WSVWM of Trusted Results
availableResults (resultHash,fileId,volId)
HashCheck[Incorrect]/
End
Send New Results to VCS
updateTrust (hashGroup,resultHash)
HashCheck[Correct]/
Waiting for hash Re-Check

Figure 12: WS Validator State Diagram

It is responsible for collecting and comparing results, and after request for the trusted file to the worker. However, all the components of the system have to work to ensure reliability, taking into account all the considerations about the whole validation process. This design implements the “m-first validation” and allows volunteers to switch dynamically between jobs without keeping them idle or requesting files continuously.

As can be seen in Figure 12, when an instance of the Validator is created by the Volunteer Control Server, it has a group of volunteer workers attached and a Volunteer Worker Manager to send the result. It begins in an idle state waiting for the hash of the different results sent by the volunteers. Once a hash is received, the Validator checks if the Volunteer was included in its group and not blacklisted; then stores it and adds it to the correspondent group of matching hashes. If that group does not exist, it creates a new one.

Then, it checks if the group has reached the number of m members. In the positive case, the Validator will ask to the waiting volunteer to wait for file request and send the

Worker Surrogate the information about this result. To finish this case, the Validator will check the available hash results, store them in a database and tell the Volunteer Control Server which volunteers sent wrong results.

In the negative case, the Validator will let the Volunteer disconnect and go to the Worker Pool as idle. If there are not enough hashes waiting to be received to reach “m”, the Validator will notify the Volunteer Control server, asking for a number of new workers. However, if it still has volunteers working to send a hash, it will come back to its first state.

4.2.4. Volunteer Worker

The Volunteer Worker is designed as two separate entities that have different responsibilities but collaborate on the volunteer's host.

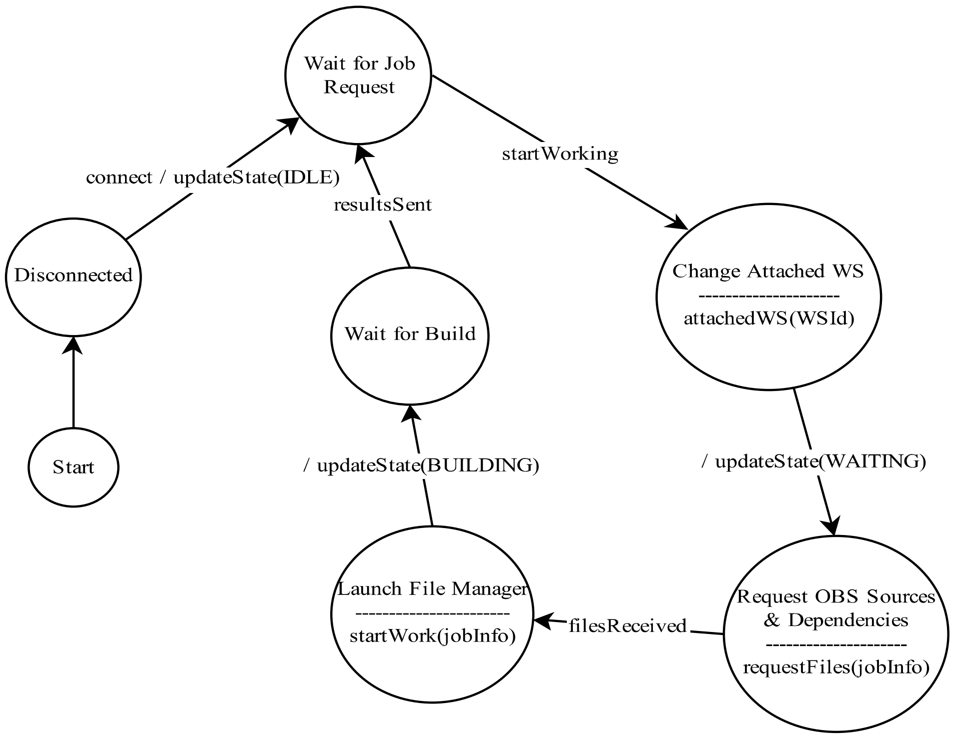


Figure 13: VW State Manager State Machine

As can be seen in Figure 13, the State Manager is the part of the architecture dedicated to maintain updated the communication with the Volunteer Control Server. It takes care of the register into the system, so the volunteer can decide to connect the application and start looking for jobs. Then it monitors the File Manager's building process keeping its state up-to-date.

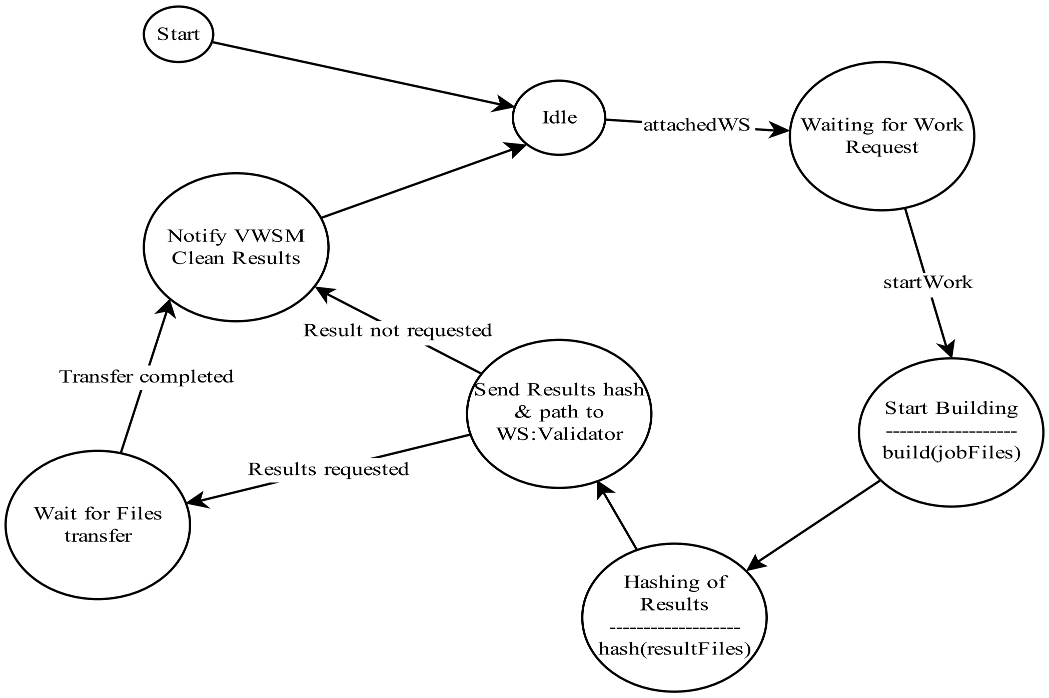


Figure 14: VW File Manager State Diagram

As can be seen in Figure 14, the File Manager is the entity launched by the Volunteer Worker that actually performs the computations. It works on a pipeline way, waiting for the different communications coming from the State Manager and OBS itself, doing the build job, hashing the results and sending them back to the Validator to wait for confirmation.

4.3. Sequence Diagram of Typical Scenario

A very simple proof of how the system will work can be shown. At start, three entities of Volunteer Workers (VW01, VW02, VW03) are launched. At the same time, on the other side the Volunteer Control Server including its Worker Pool Maintainer, Scheduler Dispatcher and Worker Pool Database is also launched. OBS Server also appears on the figure.

First, the three VW State Managers get registered on the Worker Pool Database, and quickly they notify their new state as “IDLE”. At the same time, the VCS Scheduler – Dispatcher communicates to OBS Server to obtain the queue info value, and periodically checks the idle volunteers list in the database until it obtains three available volunteers.

Then, the VCS Scheduler – Dispatcher instantiates one Worker Surrogate with the three volunteers (VW01, VW02, VW03) attached in a list. After that, it sends the id of the new attached Worker Surrogate (WS01) to the volunteers. At the same time, the WS OBS Worker Manager is connecting to OBS Server, updating its state as “IDLE” and getting the new job to perform information.

At this point, the volunteers are connected as “WAITING”, prepared for the WS Volunteer Worker Manager to distribute the new dispatched job info among them. When the order “StartJob” is received, they change its state and request, at first, the worker code to OBS.

Here a situation appears, because the worker code of VW01 is not updated, and it has to reinstall it and reboot. VW02 and VW03 are at this moment requesting the build binaries and dependencies to OBS. After building the environment, they start building the job.

VW02 is the first one on sending hash results to the Validator, as “ResultHash01”. As the group size of “ResultHash01” is lower than m (2 in this case), it receives permission to discard the results and clean the environment.

At this time, VW01 has been rebooted and now works on building the job. Now VW03 just finishes its work and sends back the hash results to the Validator. The new hash, “ResultHash02” its different to the first one, so it creates a new group which, as expected, does not reach m so its discarded.

Now, only VW01 is left to send its results. VW01's hash coincides at this time to “ResultHash01”, so the group size becomes m. Now the Validator notifies the WS Volunteer Worker Manager about an available trusted result, coming from VW01 and with hash “ResultHash01”. The Volunteer Worker Manager then requests the result files to VW01 and re-checks the old and new hash. As it is correct, the file is transferred to WS OBS Worker Manager to handle the uploading to OBS Server. At the same time, VW01 is notified to discard its results, clean environment and restart.

Finally, the Validator sends VCS Worker Pool Maintainer the new trust data concerning the three results received from the Volunteer Workers to process and include it in the database.

This case of use finishes here, but in a real situation, all the Worker Surrogate instances will be destroyed, and the Volunteer Workers, after cleaning and rebooting, will be connected as “IDLE” to continue working.

5. Implementation Work

5.1. Requirements

One important part of this thesis is the demonstration that the defined design can actually be put in practice by the use of certain technologies to implement it. As the system does not have to be truly deployed to perform as a part of the OBS workflow, many different methodologies and technologies can be used.

On the requirements we are chasing the realization of a certain idea to show it can achieve a certain goal. This goal is set by certain high-level requirements that we will discuss later on.

Roughly, a “fake” OBS Server has to be launched in a machine, ready to dispatch a job. This job can be any trivial time-consuming task; for instance, the calculations of x pi's digits, counting letters in a huge document, etc. In other machine, the designed OBS Volunteer System has to be deployed, configured to communicate with the previous one over the Internet. Now, multiple Volunteer Workers could be launched in different machines connected to the Internet, specifying them the address of our OBS Volunteer System.

Once the first Volunteer is connected, the process has to start, and after different steps depending on the current level of implementation, the task has to be performed and the results sent back to the OBS Server.

At start, the application has to be focused on better representing the communications between the different components than their performance. From here, the way is forward to make the proof-of-concept look on each iteration more realistic and closer to the proposed design.

5.2. Base Design

The first configuration that almost fulfills completely the requirements allows launching in three different machines connected over the Internet a “fake” OBS Server instance, an OBS Volunteer System instance and a Volunteer Worker application. A predefined job is dispatched by OBS Server and performed by the Volunteer Worker. Neither fault-tolerance nor fault-simulation is set by the moment.

Despite this system still seems somewhat faraway from minimum acceptable requirements, it has been designed with scalability in mind. More Volunteer Workers could be launched and attached to the system by adjusting the required group size value. Also computing faults could already be simulated and validation started by modifying the so-called “m” value. Moreover, the design is ready to maintain new groups of Volunteers attached to new Worker Surrogates.

The technologies have been selected with the constraints on mind of making the deployment as simple and lightweight as possible. The less possible external components have to be used and as a tool for a proof-of-concept, the language has to be flexible and little time-consuming, because we are interested in technology exploring rather than full-scale deployment.

As programming language, Python 2[17] has been chosen because of flexibility and low requirements, perfect to be deployed in test environments. As external dependencies, the communication is handled by Bottle's webserver, a one-file lightweight library[18], and the Requests[19] library module.

Python scripts combined with Object-Oriented-like methodologies and multithreading are the chosen techniques for the main functionality of the system. On the communication aspect, the use of webservers and HTTP requests allows bidirectional interaction.

The Bottle's webserver is designed with Restful-like methodologies [20] in mind. It is resource-oriented, using parameters on the bodies of GET and PUT HTTP requests to call functions, transmitting the information by JSON [21] data.

The class structure of the system follows the ideas of the general process design shown above. However, some new components needed to be added and others changed their responsibilities a bit because of the differences and restrictions of the communication protocols selected.

The three applications of the system are completely independent of themselves on class-level talking. This has the advantage of making possible the improvement of any of them by only respecting the communication protocols, isolated on respective “httpClient” and “httpServer” classes and modules. As can be seen, each service owns the previous expected components plus the required communication and coordination modules.

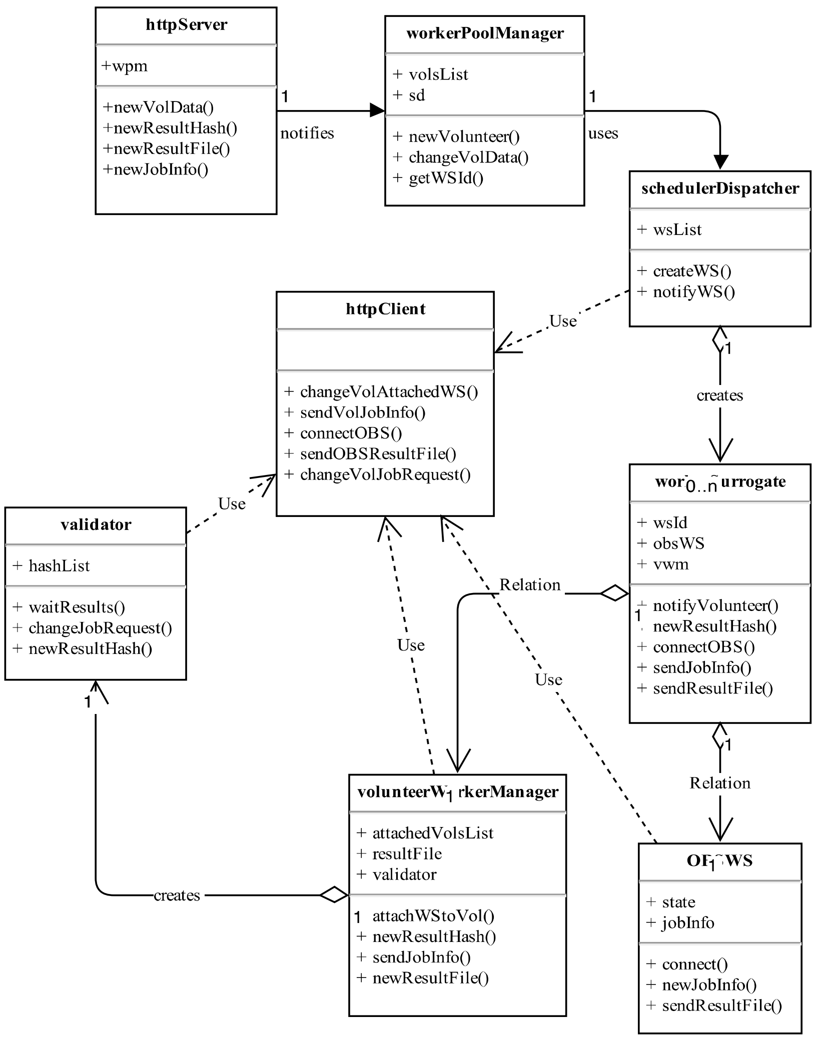


Figure 15: OBS Volunteer System Class Diagram

As can be seen in Figure 15, the OBS Volunteer System has seven classes that we could group on three levels: the Volunteer Control Server, the Worker Surrogate, and the Communications level.

On Volunteer Control Server we can find the main classes that stay always launched when the system is deployed; these are, the workerPoolManager, and the schedulerDispatcher. They are responsible for maintaining the attached volunteer's data, deploying new Worker Surrogate level instances and redirecting the communications coming from the httpServer.

On Worker Surrogate level we can find four classes. These are responsible for acting as proxy between the OBS Server and the groups of Volunteer Workers, maintaining the correct interaction protocol with the first one and managing the performance of the tasks by the second ones.

The Communications level is the main addition to the original component structure. An httpServer class is working here for each OBS Volunteer System deployment, taking care of all input data coming from out of the system. Its labor is to notify and redirect all of them to the workerPoolManager. For the output data, a httpClient module has been implemented to easily handle all the outgoing HTTP requests.

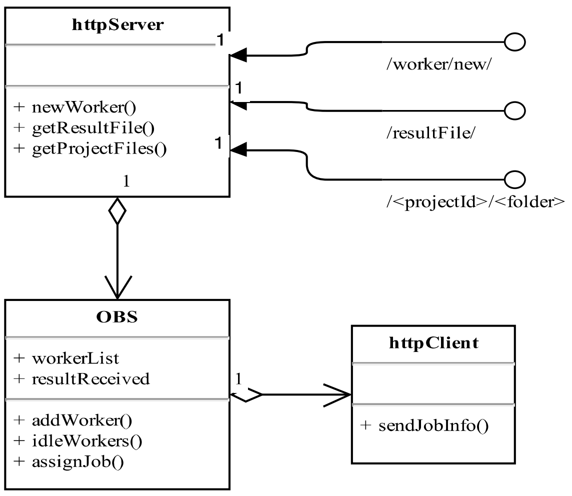


Figure 16: OBS Server Class Diagram

As can be seen in Figure 16, the “fake” OBS Server is a very simple three-class service, composed by httpServer, OBS and httpClient. Server and Client work as usual, the first one handling incoming requests and notifying the OBS kernel, and the second one preparing and managing the HTTP requests. The OBS class, is currently implemented to maintain a worker list, dispatch a predefined job and wait for the results.

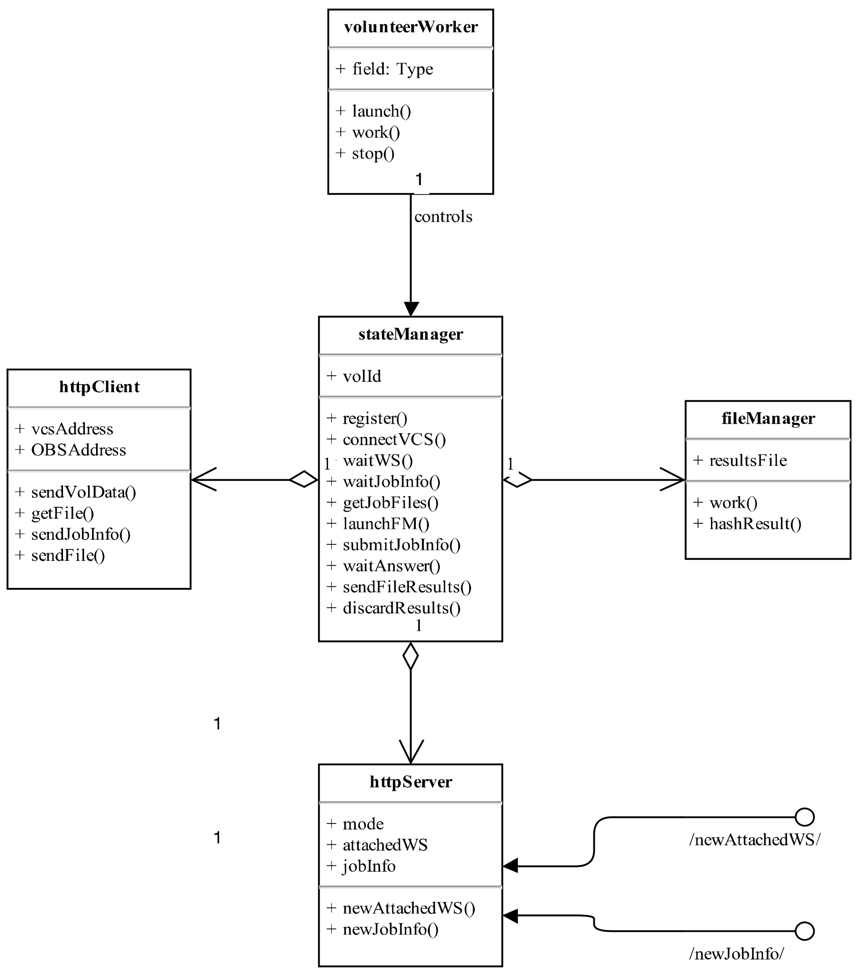


Figure 17: Volunteer Worker Class Diagram

The Volunteer Worker class diagram shows in Figure 17 five classes that can be grouped on two levels: the main level, and the communications level.

On the main level, a volunteerWorker class controls the workflow of the application and interacts with the user. The stateManager is the core of the application because it actually performs and handles the different actions of the worker. Finally, a fileManager class was implemented to ease changes on the worker if the tasks to perform change.

The communications level is composed, as always, by a httpServer class and a http-Client class that perform the expected operations.

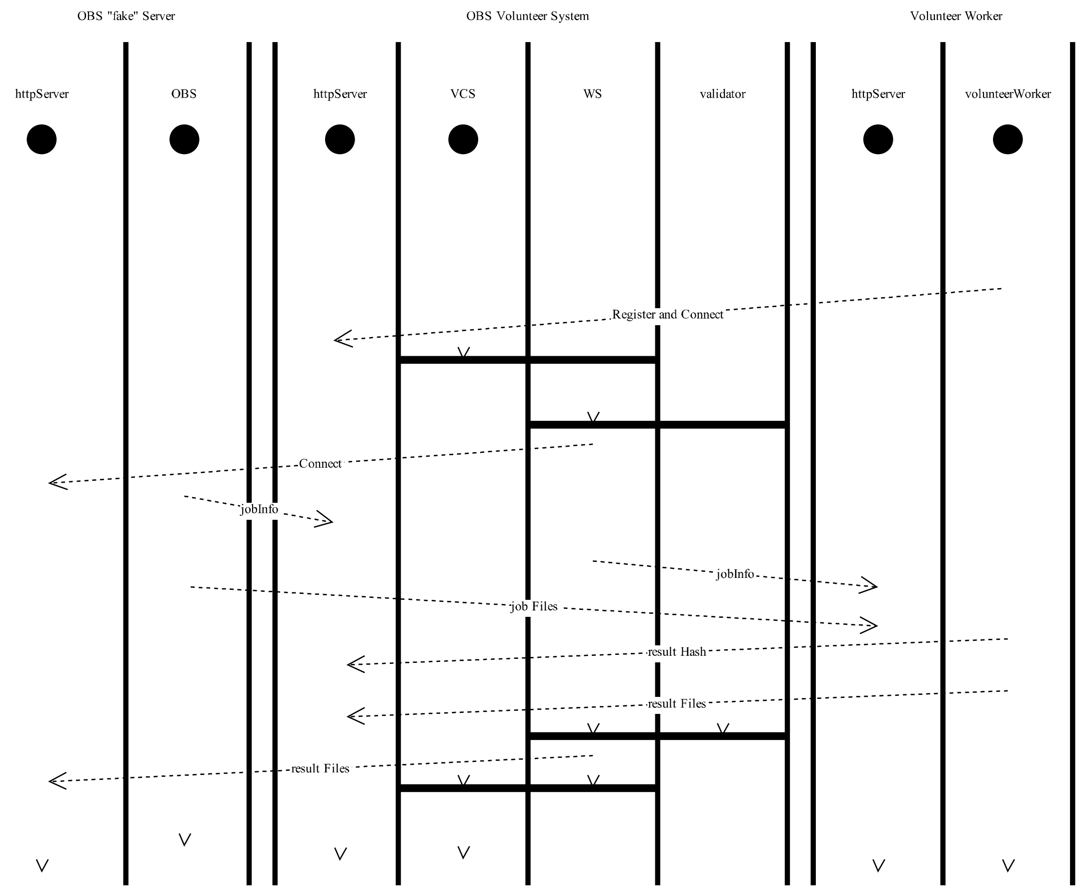


Figure 18: Threading Diagram and their trigger events

Typically, all the distributed applications that require bidirectional communication are designed following different multi-threading or multi-processing principles. By default, the communications level(usually server and client) run as a separate thread of the main body. It is done on this way because the main application flow should not care about the state of the network, and only be interrupted from that level when it is required.

Following this approach, all the HTTP servers run on different threads as can be seen in Figure 18. Moreover, based on the original design of the system, the Volunteer Control Server side is running apart from each Worker Surrogate. The Activity Diagram shown may not be representative of a real-case, but has to be considered that if the number of volunteers and jobs dispatched is increased, creating and dispatching jobs to new groups of volunteers should not be held while waiting computations to be completed. Also, to increase performance, a Validator instance is running on a different thread, so the main Worker Surrogate does not have to take care of receiving and comparing results instead of managing its attached volunteers.

Might be questioned that as a proof-of-concept, real deployment cases do not have to be taken into account, but it has been considered that at least simulating the same entities as in a real case needs to be done.

The Python-Bottle webserver following Restful-like methodologies goes as follows. On these tables, Table 1, Table 2 and Table 3, METHOD, URL and DATA packed as JSON objects into the request bodies are described. It is supposed that the respective HTTP clients have been implemented following this using the “requests” library of Python.

Table 1: OBS HTTP Server

METHOD	URL	DATA
PUT	/worker/new/	takes request address
PUT	/resultFile/	file body
GET	/<projectId>/<folder>/	file body

Table 2: OBS Volunteer System HTTP Server

METHOD	URL	DATA
PUT	/volData/new/	volId, Address, State, AttachedWSId
PUT	/volData/	volId, Address, State, AttachedWSId
PUT	/resultHash/	VolId, hashResult, jobId, requested Flag
PUT	/resultFile/<volId>/	file body
PUT	/OBSjobInfo/new/	JobInfo Dict

Table 3: Volunteer Worker HTTP Server

METHOD	URL	DATA
PUT	/newAttachedWS/	attachedWSId
PUT	/newJobInfo/	JobInfo Dict

Like can be seen, all the relevant data is included in the body of the request, and the URL is only used to refer to existing resources on memory or to flag different behaviors(as is done with “new”).

The OBS Volunteer System server handles requests both from volunteers and OBS Server. The pair “volId” – “request address” is used to check procedence of data, what is considered enough at this level.

5.3. First Iteration

This version is the first one that fulfills completely the requirements of the proof-of-concept following the specified design. However, some performance improvements like the flexible-volunteer-group-size have not been implemented yet.

At first, the system is at this stage prepared for launching it on a real-environment by any user with minimum knowledge. The three entities(OBS Server, OBS Volunteer System and Volunteer Worker) can be launched using command-line instructions and finish their execution normally. Also, they now perform the typical work flow; after performing the tasks, information is saved, environment cleaned and they come back to their start point.

Following this, the logging and user interfacing aspect has been improved: the entities now display on screen the state of computations and a log file is set to keep track of all connections and jobs performed by the volunteers.

As expected from the base design, now the validation process is fully operational. The size of the groups of volunteers “n” and the “m” number of similar hash results received can be now chosen before launching the system. The result file hash re-check(after accepting it and requesting the file to the volunteer) is active too.

To test that, Volunteer Workers can be now launched from the command-line simulating different erroneous or malicious behaviors. The working delay can be chosen, as the chances of performing a bad work and accepted file result spoofing.

Figure 19 shows an example of system start using six different terminals in the same host. The values of group size and “m-similar” results have been set to 4 and 2 respectively.

```
T1: $ python OBSServer.py
T2: $ python runWS.py http://localhost:8082
T3: $ python volunteerWorker.py http://localhost:8080 8081 -d 2
T4: $ python volunteerWorker.py http://localhost:8080 8083 -d 10
                                                          -we 100
T5: $ python volunteerWorker.py http://localhost:8080 8084 -d 20 -wf
T6: $ python volunteerWorker.py http://localhost:8080 8085 -d 30
```

Figure 19: Multi-terminal example of system's launch.

Terminal 1 runs a “fake” OBSServer instance. By default, it listens on “localhost:8082”. Terminal 2 runs the system itself. It has been specified to connect to the OBSServer instance listening on “localhost:8082”. Terminals 3, 4, 5 and 6 run four different Volunteer Workers. All of them are specified to connect to the OBS Volunteer System listening on “localhost:8080” and, as they are launched in the same machine, four different listening ports have to be chosen. Apart from that: T3 worker will spend two seconds working and send a correct result. T4 worker will spend ten seconds working and make mistakes on each data calculated 100% of the times. T5 worker will spend twenty seconds working and its hash will be correct, but it will try to send a “fake” file as result. T6 worker will spend thirty seconds working, but its result will be correct.

As the group size is set to four, once all the volunteers are connected the Worker

Surrogate will be launched and the job dispatched. As “m” is set to two, the worker on T5 will apparently send a correct result, but its file will be rejected after re-hashing. The system will have to wait for the worker in T6 to send its result to validate it.

5.4. Distributed Environment Deployment

The environment chosen to test the performance of this proof-of-concept is PlanetLab. It is a global network of computers available as a testbed for computer networking and distributed systems research. Each research project runs a "slice", that gives experimenters access to a virtual machine on each node attached to that slice. Accounts are available to persons affiliated with corporations and universities that host PlanetLab nodes. [22]

The most simple access to each node is done via SSH authentication using a terminal. By default, once connected, each remote virtual machine runs a Linux distribution with the basic packages, which is enough for our purposes. The files of the system only have to be uploaded using SCP.

Before explaining the deployment process, some changes on the codes need to be noticed. At first, the group size and the number “m” of the validation for the Worker Surrogate can be set from the command-line. At second, the HTTP servers running on the applications now detect automatically the external address of the host, avoiding one-by-one manual configuration.

To start, two nodes are attached to the slice. One will host the OBS Server, and the other one the OBS Volunteer System itself. The deployment is done manually.

For the second part, a script that auto-deploys and launches the Volunteer Workers on the nodes is used. First, it needs a file containing a list of the attached nodes to host Volunteer Workers. Second, it uploads over SCP the Volunteer Worker files to each node using the RSA key on disk to authenticate. Third, it launches on different processes the Volunteer Workers via SSH connected to a specified vcsAddress using the different stated options. This volunteers will perform their computations normally and the script will finish their execution and kill their processes when demanded.

As the script allows launching a variable number of Volunteer Workers with different options once the nodes are correctly attached and authentication accepted, many tests can be done. We left this for the evaluation part.

6. Evaluation and Analysis

6.1. Proof of Concept Tests

As stated on the Distributed Environment Deployment part, we are carrying on the tests using several nodes on PlanetLab running scripts. For these tests, we deployed one OBS Server node, one OBS Volunteer System node and ten Volunteer Worker nodes. The first and second ones are run directly from the SSH console using the commands shown on the Figure 20 with different parameters. Value “X” refers to the Group Size for each Worker Surrogate, and value “Y” refers to the “m” value for validation.

```
T1: $ python OBSServer.py
T2: $ python runWS.py [OBSServerAddress] -gs [X] -m [Y]
```

Figure 20: Console commands

The Volunteer Workers are launched from different processes of a script. This script executes the order seen in Figure 21.

```
$ ssh -l dcetut_Volunteer_Computing -i ~/.ssh/sshkey
[nodeAddress] python vol1/volunteerWorker.py [vcsAd-
dress] 8080 -d [delay] -we [failure]
```

Figure 21: Console commands

The parameter *nodeAddress* corresponds in each process to a node name from a list. The parameters *delay and failure* are taken from a data structure depending on the scenario we want to test.

The base case shown in Table 4 launches the ten volunteers without errors but having different delays in their working times. These delays are chosen from 30 to 120 seconds on a normal distribution–like way. This simulates the variety of processing power of the attached machines. The group size is set to 5, and “m” is equal to 3. This means that only half of the connected machines will actually perform the job.

After ten executions, the media of the computations is around 80 seconds, what suits completely with the central value of the delays set. This worked as expected, because regarding the group size and m value selected, we are avoiding the slowest processors, but also making wait the fastest ones.

Table 4: Base case test results

Node	Delay	Failure
planetlab4.hiit.fi	30s	0%
planck227ple.test.ibbt.be	50s	0%
planetlab2.thlab.net	50s	0%

planetlab1.ci.pwr.wroc.pl	75s	0%
planetlab1.unineuchatel.ch	75s	0%
planetlab-1.ing.unimo.it	75s	0%
plab2.ple.silweb.pl	75s	0%
planet2.elte.hu	100s	0%
pl1.bell-labs.fr	100s	0%
dplanet1.uoc.edu	120s	0%

The normal scenario includes, as seen on the table, a reasonable error ratio for each node. Exactly, a 10% failure ratio means that each Volunteer Worker will produce errors on its calculations one out of ten times. We consider this number an upper bound when talking about typical errors on processing or transmission, not caused by anomalous situations. Then, selecting the group size as 7 and m as 4, the final result obtained in OBS Server was correct ten out of ten times.

As Table 5 shows, some more tests were run also looking for abnormal scenarios. Shifting the barrier of the 50% failure ratio, ensuring a conclusive result from the first 7 workers becomes difficult. However, this does not mean that an untrusted result is sent back to OBS Server even with 90% of failure ratio, because the probability of various nodes making the same mistake is very low. In addition, reducing m to 2 over 7 on this scenario(which has the same result as adding more volunteers) allows us to obtain trusted results.

Table 5: Different error ratios tests

Node	Delay	Failure 1	Failure 2	Failure 3	Failure 3 m=2
planetlab4.hiit.fi	0s	10%	50%	90%	90%
planck227ple.test.ibbt.be	0s	10%	50%	90%	90%
planetlab2.thlab.net	0s	10%	50%	90%	90%
planetlab1.ci.pwr.wroc.pl	0s	10%	50%	90%	90%
planetlab1.unineuchatel.ch	0s	10%	50%	90%	90%
planetlab-1.ing.unimo.it	0s	10%	50%	90%	90%
plab2.ple.silweb.pl	0s	10%	50%	90%	90%
planet2.elte.hu	0s	10%	50%	90%	90%
pl1.bell-labs.fr	0s	10%	50%	90%	90%
dplanet1.uoc.edu	0s	10%	50%	90%	90%
Correct Trusted Results (over 10 tests)		10	4	0	2

Incorrect Trusted Results (over 10 tests)		0	0	0	0
Inconclusive Results (over 10 tests)		0	6	10	8

6.2. Volunteer Computing Concepts Evaluation

On the Previous Research part, we described the main concepts to judge a Volunteer Computing system. Now, we are going to analyze our system, both Design and Implementation, from those points of view.

6.2.1 Accessibility

From the ease-of-use perspective, the specified design has not been focused on its distribution or deployment yet. The proof-of-concept is mainly made to show how the design may work and to run different tests. However, it runs over any usual Linux distribution, and its relatively easy to make the Volunteer Worker perform tasks from the console only knowing the VCS address to connect with.

From the security perspective, our system only acts as an intermediary between OBS and the volunteer. Therefore, the deployed software (Volunteer Worker and Job Files) is as trustful for the user as the owner of the OBS instance is. Also, all the job files are obtained directly from OBS by design.

From the user experience perspective, there is a lack of intention on both parts, design and proof-of-concept. The design is made to first present a suitable process architecture that supports volunteering for the existing OBS system. This means that how to integrate and “sell” it to possible volunteers is an issue that relies on the organization responsible of running it together with its OBS instance in the future. Also, the proof-of-concept is not directly made for the final user. Its interface and possibilities are enough for the testing purposes it is made. However, the Volunteer Worker application is not ready to be used by a non-technical volunteer. Also, the system itself needs some knowledge about the process and should be deployed only for testing purposes.

6.2.2. Applicability

The specified design is quite universal and it is not restricted to any architecture yet. This means that it follows the path established by OBS when talking about different building platforms and possibilities of workers joining or leaving the computation. The proof-of-concept implemented is focused on using as simple technologies as possible so it works on Python over Linux. However, having followed the design, it is able to distribute any kind of computation problem, what indicates its flexibility and usability for different tasks.

The best efforts have been made on the scalability and performance aspect of the design. Working with untrusted volunteers can be considered as a barrier here, but also a reason to design the system as prepared as possible to take benefit from a huge amount of workers. Many policies have been adopted to achieve that. One of them, is the direct file distribution from OBS, that improves scalability. Never keeping the volunteer waiting for an answer, improves performance. Related to that, only requesting a file when needed, improves also performance. The dynamic adjusting of the group size also makes the system more scalable, and boosts performance in many cases.

The proof-of-concept follows also some of these principles. The VCS could already instantiate Worker Surrogates at the same time if needed. The Volunteer Workers do not stay waiting for long time, and they come back to the pool as IDLE as soon as they finish. Also, from the results shown on the chapter before, a good compromise between obtaining trustful results and do not wait for the slowest workers has been achieved.

From the programmability point of view, the requirements for the project are a constraint here because of its specific purpose. From the design, this volunteer computing system is an add-on to an existing platform as OBS. Therefore, its purpose is to communicate with it and perform the requested tasks: that is only building software packages. Even if the proof-of-concept has to show that, the kind of performed jobs is set as not important. At this moment, it distributes a Python script solving a trivial program, but it could distribute a more complex script that will be executed and replicated on the same way.

6.2.3. Reliability

Build results reliability has been considered as a main requirement for the system. As a built package may be distributed over thousands of users, a wrong result sent to OBS cannot be tolerated.

If configured correctly, the specified design achieves the goal of being full fault-tolerant. The validation process incorporates as main feature the m-validation. If the group size and m values are wisely chosen, it is almost impossible that even containing the same errors, a wrong result can be validated. The only requirement is that OBS makes sure that is sending the correct sources to the volunteers. In addition, this process is improved supporting volunteers blacklisting and performing hash rechecks each time a file passes through the “untrusted” zone.

The process can also tolerate worker malfunctioning or shutdown, dynamically requesting more volunteers and holding the results until all the steps have finished.

Considering coordinated attacks and sabotage attempts, the system trusts on the random rotation of volunteers in different clusters mixed with m-validation. A high ratio of saboteurs over the total number of volunteers is needed to validate malicious results. However, the system is not designed yet to support more complex hacking techniques that suppose direct attack to the servers.

The proof of concept implementation achieves a very good fault-tolerance like can be seen on the previous tests. The m-replication protects against any scenario of random processing failures and transfer errors. Volunteer Worker shutdown or file transfer malfunction is supported also, because the process does not discard the remaining workers until the correct file is get. However, as a proof of concept, the extra improvements that could resist a small sabotage attempt or a non-random failure, have not been implemented.

6.3. Future Work

To make the system suitable for actual utilization, some improvements concerning security and user experience need to be done. There also exist some upgrades to system performance that can be proposed even being out of the scope of the thesis.

6.3.1. *Improving efficiency*

Two optimizations on this area rely on a better utilization of the acquired volunteers' information. Adjusting trust levels of Volunteer Workers using various criteria could improve the efficiency of each Workers Cluster formed. Currently, this is done only via blacklisting, and it only affects the validation process itself, not the whole working process.

However, in a Volunteer System this idea may have also its downsides. Giving priority to some workers also opens the door to malicious volunteers. Some of them could farm a good reputation to start and after that change, or even localize the best volunteers and then impersonate them. As we have said, the first commitment is to ensure reliability then improve efficiency.

Another more simple improvement, that some Volunteer Computing systems already do on their workers, is to send a test job at start to classify the capabilities of each host. This may prevent bad machines to slowdown job computations because of validation process.

6.3.2. *Improving scalability*

One of the main challenges when applying volunteer computing to OBS is how to distribute sources and dependencies among the new pool of hosts. Transferring these files can sometimes suppose a problem because of their sizes. In addition, when replicating the jobs the number of hosts requesting files to OBS may be multiplied many times. A situation could arrive that, even having volunteer processing power available, OBS cannot take profit of it because of this limitation.

The solution is, like some other Volunteer Computing Systems or even the same organizations that distribute different Linux versions do, rely on P2P traffic. The basic idea is to distribute the files only to a part of the volunteers on the group. Then the file may be replicated directly from one to another.

However, this includes some new risks to take into account. Wrong file distribution may need important hash checking measures to avoid it. Also might be important to balance correctly the own OBS distribution with the use of P2P, so this system does not affect efficiency. The majority of the commercial Internet connections provided by the ISPs are asymmetrical, so various upload hosts are needed to provide normal download speed to a host.

In addition, behind this P2P system a complete architecture to support it is needed. It is not on the scope of this thesis to design it, but some existing projects as the Torrent protocol are showing nowadays their efficiency and flexibility. This means that as we did on the existing research of Volunteer Computing, many other systems must be studied before directly implement a new one.

6.3.3. *Improving reliability*

Even if we took this feature of Volunteer Computing as our main objective on the design, there are still few more improvements that could be added.

One of them during the validation process itself, is the possibility of asking for result files on a provisional way to the volunteers even if “m” has not been reached yet. Working with big group sizes and time-consuming jobs, this may help to obtain a kind of “backup” in case the chosen volunteer shuts down.

However, which value “l” select as reasonable has to be carefully considered. Apart to be a high percent of “m”, it should take into account the accumulated values of the other hash groups. We could say that “l” is this value when regarding all the available results, you can assure by a 90% that is going to be the chosen one.

Another improvement that many volunteer computing systems consider when they try to achieve sabotage-tolerance is the so-called “spot-checking”. Here, some jobs which result is known in advance called “spotter-jobs” are dispatched from time to time to calculate the credibility of the volunteers. Some papers show methods combining m-validation and spot-checking that show systems statistically almost immune to failure and sabotage. [23, 24]

6.3.4. *Improving security*

At the required stage of the design, security on real-environment is not a concern because which technologies to use have not been specified yet. However, is good to know that to make this system have acceptance, some weak points need to be secured.

As we are offering OBS a piece out of the organization that is acting as several official workers, we need to ensure that this worker or its communications cannot be impersonated or kidnapped by attackers. On a high abstraction level we propose key authentication to know exactly what attached worker corresponds to each OBS Volunteer System instance, and virtualization to separate the build host to the rest of it could be used.

The other weak point of the system is the relation between each Volunteer Worker and the central architecture. On this phase of the work flow the executiion goes out of our servers. In addition to fault and sabotage tolerance systems, at some level the system may need to implement key authentication to identify the volunteers working on each moment.

On the other hand, we could decide to assume that all the volunteers are going to be treated on the same way because we cannot identify them correctly. This might cause that at some point on the time a high-scale coordinated attack makes our system validate malicious results.

6.3.5. *Improving usability*

In general, the acceptance of a Volunteer Computer System depends on its ability to offer striking feedback to the potential volunteers. We believe that this depends mainly on the policies of the organization running the instance to attract volunteers. Rather than imposing a front-end for the architecture by design, our way is to offer possibilities to give the users the requested feedback.

The Volunteer Worker has been designed to separate the working process from the one that monitors it. This allows any designer to implement a structure to keep the user informed about the state of the execution, like BOINC's screensaver.

The Volunteer Control Server structure is prepared to hold real-time information about running jobs and attached volunteers. This may help to deploy reputation or award systems to hook up volunteers, even if it might be harmful for the validation process.

In any case, we consider that the success of a Volunteer Computer system relies on the organization and its way to advertise and to ease joining the computations via web.

7. Conclusions

This thesis presents how package building can be distributed over many heterogeneous, untrusted and sometimes unreliable hosts belonging to unknown standard users. Specifically, it provides a solution to allow Open Build Service work full-part or hybrid as a volunteer computing project. This saves costs in processing power and structure maintenance while gains access to a huge pool of computational resources. However, it brings along new challenges regarding accessibility, applicability and reliability.

The change from an in-house system to a volunteer computing system needs to be carefully measured. As shown on the evaluation chapter, it has to fulfill certain requirements to be successful. The system has to be efficient and effective for package building and also tolerate unreliable and faulty results coming from the volunteers without affecting the final result. It is also required to ensure volunteers security and ease-of-join computations. Not accomplishing this goals may end in an effectiveness drop of OBS.

The solution of the last situation is the proposed design, as it fulfills all requirements of a volunteer computing system without implicating the actual OBS system. It can be understood as an add-on that allows an hybrid system offering OBS new workers based on volunteers' work. Different from migrating the current system to existing platforms like BOINC, it permits progressive adaptation of package building tasks to volunteer systems, using a design specifically intended to it.

The analysis of the solution shows that different to the existing systems, the process has to be and is focused on tasks that require bandwidth, processing power and storage capacity. This supposes a challenge talking about scalability and efficiency of the new system. In addition, as many researches have demonstrated, the validating system chosen evidences to be fault-tolerant even sabotage-tolerant with few more additions.

The volunteer system implemented in this work was a simplified proof-of-concept. The test measurements taken in the evaluation show how the system is efficient dispatching and collecting results from the volunteers, obtaining them in an average time compromise between speed and reliability. These tests also show how effective is the validation system exposed to random faults, achieving a situation where is impossible to accept a wrong calculation.

However, for real-environment use, design and moreover implementation have lacks talking about accessibility and security. Because of its conceptual design and proof point of view, ease-of-use and user experience have not been taken into account yet. Because of being a high-level design, neither the specific technologies nor the way to secure them has been set yet.

Therefore, the use of volunteer computing will play in the future an important role in supporting package building, as they currently do in other tasks. The exponential growth and universalization of the computing power available to the standard user manifests that following the correct path could have access to resources that they never had before.

8. References

[1] Volunteer Garage [WWW]. [Retrieved 14-9-2013] http://www.volunteer-computing.org/

[2] David P. Anderson , Kevin Reed . Celebrating Diversity in Volunteer Computing . Proceedings of the 42nd Hawaii International Conference on System Sciences – 2009 .

[3] Adam L. Beberg, Daniel L. Ensign, Guha Jayachandran, Siraj Khaliq, Vijay S. Pande . Folding@home: Lessons From Eight Years of Volunteer Distributed Computing . IEEE International Symposium on Parallel & Disitributed Processing, 2009 IPDPS

[4] Luis F.G. Sarmenta, Satoshi Hirano. Bayanihan: building and studying web-based volunteer computing systems using Java. Future Generation Computer Systems Special Issue on Metacomputing 1999

[5] Muhammad Nouman Durrani, Jawwad A. Shamsi . Volunteer computing: requirements, challenges, and solutions . Journal of Network and Computer Applications (2013), http://dx.doi.org/10.1016/j.jnca.2013.07.006i

[6] BOINC [WWW]. [Retrieved 21-09-2013] http://boinc.berkeley.edu/

[7] Christophe Cérin, Gilles Fedak . Desktop Grid Computing. Volunteer Computing and BOINC. CRC Press, Taylor & Francis Group. pp. 12 – 22

[8] XtremWeb : the Open Source Platform for Desktop Grids. [WWW]. [Retrieved 17-09-2013] http://www.xtremweb.net/

[9] Christophe Cérin, Gilles Fedak . Desktop Grid Computing. The XtremWebCH Volunteer Computing Platform. CRC Press, Taylor & Francis Group. pp. 55 – 62

[10] Baden Hughes . Building Computational Grids with Apple's Xgrid Middleware . 29th Australasian Computer Science Conference (ACSC2005), Hobart, Australia.

[11] Univa Grid MP. [WWW]. [Retrieved 25-09-2013]. http://www.univa.com/products/grid-mp.php

[12] Grid MPTM Platform Version 4.1. United Devices, Inc. pp. 1 – 15

[13] Folding@home [WWW]. [Retrieved 12-09-2013]. http://folding.stanford.edu/

[14] OpenSuse Build Service. [WWW]. [Retrieved 12-09-2013].

https://build.opensuse.org/

[15] Free Software Foundation, Inc. GNU General Public License. [WWW]. [Retrieved 02-02-2014]. http://www.gnu.org/licenses/gpl.html

[16] Open Build Service (OBS). [WWW]. [Retrieved 11-09-2013]. http://openbuildservice.org/

[17] Python v2.7.6 Documentation. [WWW]. [Retrieved 02-12-2013]. https://docs.python.org/2.7/

[18] Bottle: Python Web Framework. [WWW]. [Retrieved 02-12-2013]. http://bottlepy.org/docs/dev/index.html.

[19] Requests: HTTP for Humans. [WWW]. [Retrieved 12-12-2013]. http://docs.python-requests.org/en/latest/.

[20] REST API Tutorial. [WWW]. [Retrieved 03-12-2013]. http://www.restapitutorial.com/lessons/whatisrest.html.

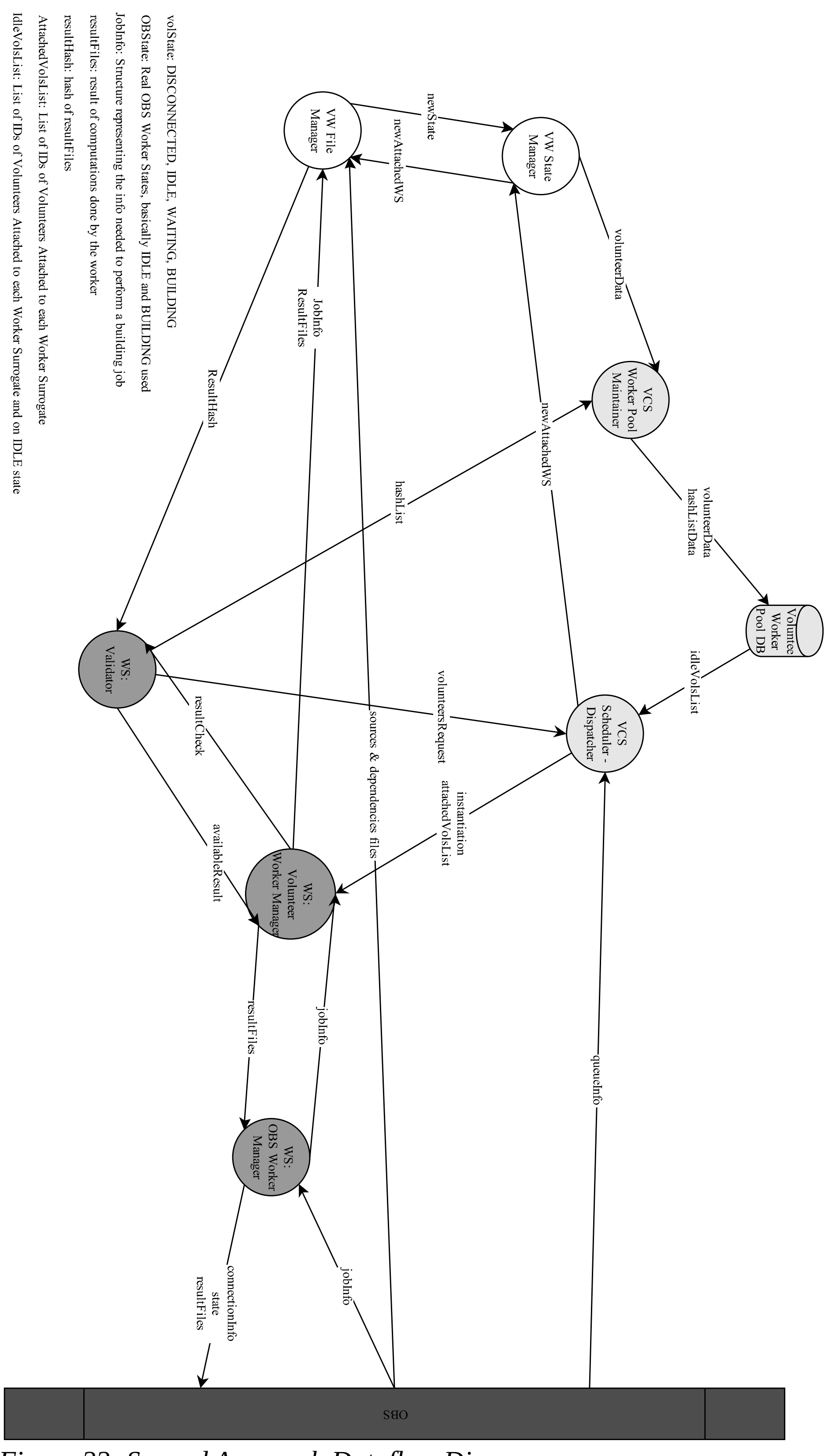
[21] JavaScript Object Notation [WWW]. [Retrieved 05-12-2013]. http://www.json.org/

[22] PlanetLab Europe. [WWW]. [Retrieved 15-03-2014]. http://www.planet-lab.eu/

[23] Watanabe, K. Fukushi, M. Generalized Spot-Checking for Sabotage-Tolerance in Volunteer Computing Systems. Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference. Melbourne, Australia.

[24] Watanabe, K. Fukushi, M. Collusion-Resistant Sabotage-Tolerance Mechanisms for Volunteer Computing Systems . 2009 IEEE International Conference on e-Business Engineering .

Appendix 1



volState: DISCONNECTED, IDLE, WAITING, BUILDING

OBState: Real OBS Worker States, basically IDLE and BUILDING used

JobInfo: Structure representing the info needed to perform a building job

resultFiles: result of computations done by the worker

resultHash: hash of resultFiles

AttachedVolsList: List of IDs of Volunteers Attached to each Worker Surrogate

idleVolsList: List of IDs of Volunteers Attached to each Worker Surrogate and on IDLE state

Figure 22: Second Approach Dataflow Diagram