# *RMrun*: peer-to-peer sharing of applications

Luís Pinto[1], Tiago Santos[1], Filipe Araujo[2]

[1] CISUC, Dept. of Informatics Engineering, University of Coimbra, Portugal
{lpinto,tjsantos}@student.dei.uc.pt
[2] CISUC, Dept. of Informatics Engineering, University of Coimbra, Portugal
filipius@dei.uc.pt

**Abstract** In this paper we describe the first steps we gave to construct a prototype of *RMrun*, a peer-to-peer system where users share programs they have with each other. We enumerate some of the problems we are facing and some directions we are considering to take. *RMrun* uses standard Unix tools, like the X protocol and Secure Shell (SSH) to run remote programs pretty much as in any Unix system. The difference is that it does not require a particular user account in server hosts. One single account enables the user to run any program available in anonymous computers inside its community.
We think that *RMrun* can fit two main trends of uses: utilization of programs by peers that do not have access to such programs or do not want to configure them; and utilization of desktop CPU time in public volunteer computer.

## 1 Introduction

In this paper we describe a peer-to-peer system, called *RMrun*, where we enhance standard services of Unix-like operating systems, to let users run programs remotely without owing an account in the server machine. We use well-known mechanisms, most noticeably, the Secure Shell (SSH) [13] and the X protocol [12]. One fundamental component of our architecture is provided by a Central Server, which is responsible for authentication, accounting and keeping track of programs available in the community[3]. With the aid of public and private keys of the SSH protocol, the Central Server makes it possible to use a single account to run programs in all the machines of the community. In this way, users can run applications they do not have, including applications installed in different operating systems or applications requiring resources beyond user's local capacity. While this is already possible with standard technology, *RMrun* reaches further, by enabling users to get applications from anonymous peers that they do not know and that may be under different administrative control without any human intervention. Put in simple terms, we want to make a peer-to-peer network where users share execution of programs and consequently CPU time.

[3] We have plans to decentralize the functionalities related to keeping and searching for information currently held by the Central Server.

We believe that one of the most interesting uses of our work is for Desktop Grid computing (also known as public volunteer computing). By enabling anonymous execution in remote computers, *RMrun* is somewhat similar to Desktop Grid systems, like BOINC [2] or XtremWeb [7], but mainly to OurGrid [4]. In BOINC and XtremWeb nodes use their idle time to compute tasks they pull from a central server. OurGrid is a little bit different, as it lets labs submit jobs to each other using a push-based approach. The main difference in *RMrun* is that we do not require the use of virtual machines to support the sandboxes needed by OurGrid. We are looking for a lighter approach, where the sandbox can run inside the operating system of the target machine. This approach also makes our system available for other uses, because all services available by default in the operating system are available for the remote running application. One of these services is the X protocol that will enable users to run interactive programs anonymously and without actually installing them. This scenario can be especially appealing for clients using non-Unix operating systems, namely for Windows users (as long as they install an X server).

Naturally, *RMrun* raises important problems, specially security, because operating systems might run untrusted programs or trusted programs interacting with untrusted users. Clearly, we must ensure that neither the client, nor the server can damage each other. Another problem is scheduling, as CPUs become freely available to third parties. Without any protection, anyone could just flood the entire system with their own jobs. Additionally, we must also ensure that each user has access to its fair share of resources inside the system. In the rest of the paper, we present the architecture of *RMrun*, we discuss problems that it raises and solutions we adopted to some of these problems. This paper should be regarded as work-in-progess, as we have not finished our system yet.

## 2 Related Work

In this section we overview the most relevant systems we are aware of, which have similarities to *RMrun*. We focus on platforms that enable users to run interactive programs. However, as we explained ahead our system also shares some goals with Desktop Grid systems, such as BOINC [2], XtremWeb [7], or OurGrid [4].

Sun™Secure Global Desktop Software[4] is an application sharing commercial software that relies on a single Data Center, where all applications are installed and maintained. Users are allowed only to run what the Central Systems has to deploy. If clients need any new or different version of software they must ask or request its installation on the Data Center. Despite being much easier to secure than *RMrun*, this software also lacks the flexibility and scalability that a peer-to-peer approach can bring. Citrix XenApp™ [5] (formerly Citrix MetaFrame Server) has server-side application support for Microsoft Windows™, Linux and Apple™'s Macintosh Operating Systems. The disadvantage of this implementation is that it exports a Terminal Server Client to provide the Operative System layer to the application

[4] `http://www.sun.com/software/products/sgd`
[5] `http://www.citrix.com/pstrans`

that the client wants to run. Moreover, the applications available are centrally managed and users must consume an operating system license in addition to the license of the software they want to use.

Another interesting and similar trend is to run and maintain the users' desktop in a remote server accessible with a particular program or even with a browser. With this approach a user can transparently change from one machine to the next and still access its own desktop space in the same state where it left. Usually, by using some common protocol like X Windows Server Protocol [12] or by running on a browser, the user can work on any operating system he or she wants. The main limitation of these systems concerns the fact that users only have access to applications existing in the central server. For instance, DOD — Desktop On Demand[6] — Online computing, uses Java technology to provide access to remote storage and some pre-defined applications. Dektoptwo[7] is similar to DOD, but the remote desktop is presented within a browser, providing seamless access across all operating systems. It shares most of the limitations observed in DOD. OpenNX[8] is the platform that most resembles our solution. It uses SSH to encrypt and exchange data and relies on an X environment to display and manage input and output. However, it does not provide a peer-to-peer approach like our does and requires the usual forms of authentication.

Finally, there are some frameworks that aim to simplify program development for Grids. Although simple and powerful as a "gridifying" approach, goals of these projects are fundamentally different from our own goals, as we want to run remote (sequential) programs exactly as they are. For instance, GridGain[9] is a software for programmers who want to develop parallel applications for Local Area Networks (LANs). To the best of our knowledge and given all the solutions we enumerated here, we think that *RMrun* has some unique features, as it is decentralized with respect to the applications that are available and it allows transparent access to resources even over different administrative domains.

## 3 Architecture of *RMrun*

### 3.1 Overview

Figure 1 illustrates the *RMrun* architecture in its simplest form. It has three components: client peer, host peer and Central Server. Client peer is the node trying to run a remote application, while the host peer is the node that offers that application. The distinction between both types of peers serves mainly for illustration purposes. Usually any computer running *RMrun* can play both roles simultaneously. The Central Server is the heart of the system as it is responsible for setting up all the connections between peers. It holds all information of interest in the system, like registered users, online users or existing applications.

[6] `https://desktopondemand.com`
[7] `http://sapotek.com/`
[8] `http://www.nomachine.com`
[9] `http://www.gridgain.com`

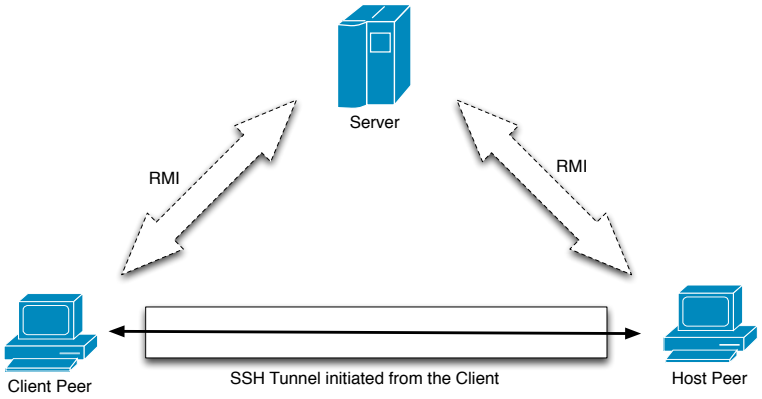


**Figure 1.** Overview of *RMrun* architecture

Typical interaction in the system takes place in four steps: *i*) each user registers itself; *ii*) host peer gives a list of its application to the Central Server; *iii*) client peer searches for an application and starts it; *iv*) with the help of the Central Server, peers establish a direct connection and start interaction.

### 3.2 Central Server

In the current version of *RMrun* the Central Server plays a fundamental goal. Although we intend to distribute some of its functions in the future to increase scalability of the system, functions like authentication and accounting are much easier and probably more reliable (in the sense that they are easier to control) if we use a centralized approach, or at least some centrally controlled hierarchy of servers. However, we will not follow this latter approach in the current state of *RMrun* and we stick to the simpler approach of having a single Central Server.

Communication between client peers and Central Server uses Java Remote Method Invocation [9] (RMI). When peers enter the network, they have to authenticate in the central server. Then, at any point in time, if the user wants to share an application, he or she has to register this application in the Central Server. On the other hand, to know which applications exist in the community, clients periodically pool the Central Server to request that information. As soon as any peer finds out about the new available application it can start using it. One issue that we can discuss here regards the best architecture to inform peers about available applications. While pooling may have some scalability problems, using something like a callback interface in the client peers was not considered for the time being, because peers can sit behind network address translation schemes (NATs) or firewalls (as we discuss in Section 5). Also, this would only work for small networks with few users and applications, as we cannot afford to block the Central Server for the time it is sending notifications to applications and then wait-

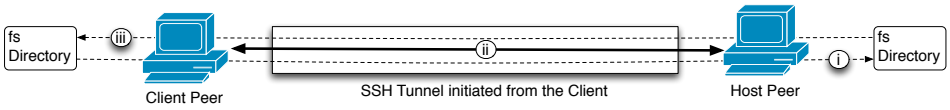


**Figure 2.** Interaction between local and remote `fs` directories

ing for confirmations. A possibility here is to use publish-subscribe [6] techniques, like the one-way CORBA invocation [5], for instance.

As we wrote before, one of the most fundamental characteristics of *RMrun* is that it allows users to enter in each other's machines without needing a password for each machine. To do this we use a special user account. The scripts that install *RMrun* create a special user, named `bic`[10]. All SSH connections use this account. In the target `bic` account we create a directory called `fs` that serves as the remote repository for the files needed by the client peer. In the current state of *RMrun*, all the local files that reside in directory `fs` are copied to the same directory in the remote `bic` account. Our system takes care of updating the local contents as we describe ahead.

### 3.3 Connection of Peers

When a user wants to run a remote application, *RMrun* client peer will request information of the host peer location from the Central Server. In response it gets the IP address of the host peer. It is important to notice that the client peer does not have any control on *where* is it going to run the application, as only the Central Server is responsible for such decision.

During connection, peers must go through three phases: *i*) upload the local directory to the remote machine (RMput); *ii*) run the application (RMrun) and *iii*) periodically download the directory (RMget). Before the RMput phase initiation, the Central Server must put the client public key in the host peer. In the RMput phase, files that are in the local directory used by the application are copied to the host peer using the Secure Copy[11] (SCP) utility. This enables manipulation of these files by the remote application, during the RMrun phase. In the transition from RMput to RMrun phases, we must add the application name to the client public key header. This ensures that any connection using the Secure Shell [13] (SSH) protocol will only allow execution of the application specified in the header. All interaction during RMrun phase takes place using the SSH and the X Window System Protocol [12] forwarded through the SSH connection (shown in Figure 2). We still need a third phase, the RMget. Has before, to initiate this phase, the key must be changed to a simple client public key. The RMget phase allows to retrieve the files back to the client's local directory. This is made on a periodical basis until the connection is broken or when the application is closed. This interaction

[10] This name derives from the name of the scholarship that supports researchers involved in this project: "Bolsa de Iniciação Científica".

[11] `http://support.real-time.com/linux/web/scp.html`

is illustrated in Figure 2. One thing to notice is that although the communication is made over secure channels, no clear text passwords are ever sent.

Currently during the several stages of tunnel establishment and file system copy back and forth, we change the client public key header to ensure that only the command to be executed can, in fact, be run, thus avoiding malicious code or instruction execution even by third parties.

### 3.4 Resource accounting in *RMrun* GRID

One part of *RMrun* that remains to be done and that is of utmost importance is the accounting mechanism. Accounting will keep track of all the resources used by each peer in the system. To do this, the client peer has to pay some "virtual credits" to the host peer in order to run programs there. To simplify the management of this system, it is up to the Central Server to transfer credits from one peer to the other (in fact, this operation never leaves the Central Server as all the information is kept there). In a first approach, we will use a simple taxation based on time of use, but we believe that the use of a credit-based system, enables the use of arbitrarily complex markets (e.g. [1]) that can fully exploit all the resources available in the *RMrun* community. With a successful market, we should not need to run a scheduler to order accesses to available resources, as the adjustment of prices should take care of that. Payment for each job submission, together with some standard tools that force user intervention at registration time (e.g., identification of letters in a drawing) should also reduce the level of threat coming from mischievous users trying to make the system unusable. For instance, anyone willing to carry a denial of service attack would first need to subscribe a very large number of users in the system. And to be successful it would first need to serve the *RMrun* community for some time (to get payment credits).

## 4 Security

We are aware that our system raises significant security problems. However, most if not all of these problems have been studied before and therefore, we can use or adapt some existing solutions to our own architecture. Basically, *RMrun* can cause security problems both to the client that will write contents sent by the host peer, but also to the host peer itself, as it will be under control of the client.

### 4.1 Hosted Application Certification

The clients only execute whatever each host deploys as available, but due to the universe of open source applications in existence, there is no easy way of checking or asserting the reliability of a particular binary object about to be run. Even by an MD5 check schema, the same application on different environments would lead to different, yet valid, results. Verifying hosted applications would lead to an immense overhead involving candidate binary objects submissions, denying use of dynamic linked libraries, building a huge database for each of these specifications

and corresponding operating systems flavours. Another completely different approach would be to give reputation scores to peers. To minimize the danger of running uncontrolled software, we limit the actions of such (remote) software to the (local) `fs` directory.

### 4.2 Running Untrusted Applications in the Host Peer

Although the host peer can limit what the client can run, behavior of some applications can be basically unknown, which means that we cannot trust them. A usual way to solve this problem is to use sandboxes, such as Xen [3] (as in OurGrid), jails or chroot. Unfortunately, none of these sandboxes fits our needs, as we do not want to install a second operating system in user's machines and we need to let users access relatively arbitrary positions in the file system to run binaries or code inside shared libraries. This problem was studied before. For instance, in [10], Peterson defines a sandbox system call for Linux. In our work, we intend to follow these steps, as we think that this is the level at which we need to protect the system.

### 4.3 Communication

Another sort of problem concerns the existence of eavesdroppers in the system that might be listening to some part of the communication that is ongoing in clear text as it is the case of RMI. To overcome this problem, we can use RMI over SSH communications using, for instance, RMISSH[12].

## 5 Open Issues and Roadmap

### 5.1 Implementation Issues

In this Section, we review some implementation problems of our architecture and we point out some possible solutions that we are considering to take in the near future. In first place, in this version of *RMrun*, we were forced to do periodic downloads of the remote `fs` directory, because we were unable to use "libfuse"[13]. Our first option was to use libfuse to map the local `fs` directory in the remote `bic` account, such that all the utilization of this directory would be in fact automatically done in the local, client peer, instead of the remote host peer (although it seemed as an operation on the local file system to the remote running application). However, libfuse is not available in operating systems like Mac OS X™ and it requires installation of kernel modules in Linux, something that we want to avoid. Despite the fact that we are considering some improvements to the `fs` directory scheme in the `bic` account, this problem remains largely unsolved, because we are not considering to re-implement or port libfuse.

Another problem that we still need to solve is the well-known problem of reaching peers sitting behind Network Address Translation (NAT) schemes or Firewalls

[12] `http://docs.huihoo.com/proactive/3.2.1/SSHTunneling.html`
[13] `http://fuse.sourceforge.net/`

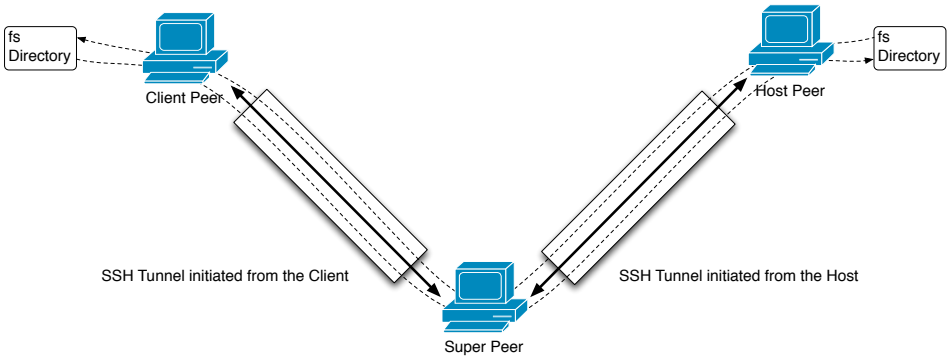


**Figure 3.** Overview of *RMrun* with super-peers

(see for instance [11]). In this case, communication between them must occur through an intermediate node with connectivity from outside. We intend to use an intermediate node with a public IP address and high capacity to serve as a tunnel to enable communication between isolated nodes. Both, the client peer and the host peer can use this scheme to traverse firewalls and NAT schemes. At the present time, we are implementing a scheme that uses SSH remote port forwarding to establish a tunnel to the host peer. This situation is illustrated in Figure 3.

The existence of multiple types of operating systems also raises a problem that limits the construction of this architecture, as some of these usually do not have any installation of the SSH protocol (e.g., Microsoft Windows$^{TM}$). To run *RMrun* even as a client in a Microsoft Windows$^{TM}$ machine we need to use an OpenSSH[14] software provided either by the X Windows package or by a separate one. To overcome this problem, we intend to resort to a solution that directly supports OpenSSH in the Java Technology libraries. In this way we surpass the inconvenient of having to install a possible add-on package for that sole purpose, with all the overheads that come with this. Additionally, we also discard the need to use system calls to provide the secure tunnelling operations. A library like SSHTools[15] will give OpenSSH encryption functions and operational security within *RMrun* itself, ensuring more modularity to the code, more Operating System independence and push even further away the chances of users accidentally changing private or public keys rendering the current application session useless. Even a system administrator could only stop *RMrun* from running, but could not tamper its generated keys or commands. Both the client, host and super-peer `bic` accounts will no longer need OpenSSH keys to log into the the peers. The user `bic` account will no longer need execution privileges nor initialization files, as it will serve for storage and file copying only.

[14] `http://www.openssh.com/`
[15] `http://sourceforge.net/projects/sshtools`

### 5.2 *RMrun* as a Volunteer Computing Platform

It is a well-know fact that some of the most powerful computing platforms on Earth are formed by volunteers running projects based on the BOINC middleware [2] (e.g., SETI@home, Rosetta@home, Climateprediction.net, among many others). Other public and commercial volunteer computing platforms include XtremWeb [7] or Google effort [8]. With *RMrun* we intend to reverse the usual approach of Volunteer Computing. Instead of following a pull-based approach, as in BOINC or XtremWeb, we want to let projects submit tasks on available resources. This means that, while BOINC clients request tasks from the central supervisor whenever they are free to compute new ones, we want to follow the reverse approach of letting the server of a project submitting tasks in computers that are available. To some extent this is similar to OurGrid [4]. However, we intend to follow a different approach: we want to do sand-boxing inside the operating system as tasks should run as ordinary *RMrun* remote processes. OurGrid uses the Xen virtual machine [3] and thus runs an entire operating system. Another difference is that we want to go beyond the tit-for-tat mechanism as we account resource usage in the Central Server. This makes it possible for a popular project to receive donations from peers, thus being allowed to run and, at the same time, it also enables a network of favors where A lets B run a program, while B lets C and C lets A, in a cycle that can have any length. In fact, since we use "virtual" credits, these can be used exactly as wished by users in unpredicted ways.

## 6 Conclusions

In this paper we presented *RMrun*, which is a software that enables users to run remote applications in machines where they do not have accounts. *RMrun* uses standard protocols like SSH and X running in end user's machines and is backed by a dedicate central server for authentication, accounting and listing of available applications. We foresee the use of *RMrun* either to run applications that do not exist on the local machine (or operating system), but also to serve as the base for volunteer computing.

While we have a running prototype of *RMrun*, we still have a long way to go, before we can make our work publicly available. Although we still need to improve some aspects of the architecture, we believe that the main issue we face is security. As a consequence, our next steps will be devoted to work on sand-boxing solutions capable of preventing or reducing the impact that unknown users can have when they run untrusted applications.

## Acknowledgements

This work was partially supported by Fundação para a Ciência e a Tecnologia under the project GRID/GRI/81727/2006, "GRID para simulação e análise de dados de ATLAS/LHC".

## References


1. David Abramson, Rajkumar Buyya, and Jonathan Giddy. A computational economy for grid computing and its implementation in the nimrod-g resource broker. *Future Gener. Comput. Syst.*, 18(8):1061–1074, 2002.
2. Dave Anderson. BOINC: A system for public-resource computing and storage. In *5th IEEE/ACM International Workshop on Grid Computing*, Pittsburgh, USA, 2004.
3. Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, New York, NY, USA, 2003. ACM Press.
4. Walfredo Cirne, Francisco Brasileiro, Nazareno Andrade, Lauro Costa, Alisson Andrade, Reynaldo Novaes, and Miranda Mowbray. Labs of the world, unite!!! *Journal of Grid Computing*, 4(3):225–246, 2006.
5. The common object request broker: Core specification, 2002.
6. Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, 2003.
7. Gilles Fedak, C. Germain, V. Neri, and Franck Cappello. Xtremweb: A generic global computing system. In *1st Int'l Symposium on Cluster Computing and the Grid (CCGRID'01)*, pages 582–587, Brisbane, 2001.
8. The google compute project.
9. Java$^{TM}$ remote method invocation specification, 2004. Revision 1.10, JavaTM 2 SDK, Standard Edition, v1.5.0.
10. David S. Peterson. A flexible containment mechanism for executing untrusted code. Master's thesis, University of California, Davis, 2003.
11. J. Rosenberg, J. Weinberger, and C. Huitema. Stun - simple traversal of user datagram protocol (udp) through network address translators (nats). Request for Comments 3489, March 2003.
12. Robert W. Scheifler. X window system protocol, version 11: Alpha update. Request for Comments 1013, April 1987.
13. T. Ylonen and C. Lonvick. The secure shell (ssh) protocol architecture. Request for Comments 4251, January 2006.