

# GiGi: An Ocean of *Gridlets* on a “Grid-for-the-Masses”

Luís Veiga      Rodrigo Rodrigues      Paulo Ferreira  
INESC-ID/IST

Rua Alves Redol N° 9, 1000-029 Lisboa, Portugal  
{luis.veiga,rodrigo.rodrigues,paulo.ferreira}@inesc-id.pt

## Abstract

*There have been a few proposals aiming at bridging the gap between institutional grid infrastructures (e.g., Globus-based), popular cycle-sharing applications (e.g., SETI@home), and massively used decentralized P2P file-sharing applications. Nonetheless, no such infrastructure was ever successful in allowing, in a large-scale, home users to run popular desktop applications faster, by using spare cycles in other users’ machines and, in return, donate their spare cycles to run other users’ applications.*

*We present a novel application and programming model that was designed to overcome some of the barriers to the deployment of a generic peer-to-peer grid infrastructure. In particular, we want to enable a trivial deployment in such infrastructures of existing applications that are in widespread use but do not currently exploit parallelism for improved performance.*

*The model presented in this paper revolves around the concept of a Gridlet, a semantics-aware unit of workload division and computation off-load. A gridlet is a chunk of data associated with the operations to be performed on the data, and in many cases these operations consist of unmodified application binaries. Moreover, the concept of gridlet is also employed for resource management, and accounting of peer contribution.*

*We believe this new concept, absent in other proposals, will significantly lower the barriers for exploiting parallel execution in popular applications, thus improving the chances of the gridlet model being widely adopted.*

## 1 Introduction

The most visible use for Grid computing today is to enable the speedup of applications that require massive amounts of computation, e.g., applications for scientific research, drug discovery, or financial risk analysis. These applications run on a Grid infrastructure that usually consists of the interconnection of clusters of well-managed machines, dedicated to implementing the Grid infrastructure. In such a virtual organization, the machines in each cluster are under the same administrative control, and access to machines across different clusters must be negotiated in advance.

For approximately a decade now, distributed cycle-sharing has been in widespread use. This has begun with the introduction of applications such as SETI@home [4], and has been followed by large number of recent others, dedicated to climate prediction [2], protein folding [12]

for drug research, detection of gravitational waves, detection of prime numbers, celestial bodies, etc. These applications traditionally follow a rigid client-server model, with a centralized server. Clients, normally run as screen savers in user machines, fetching blocks of data from the central server, to perform CPU-intensive calculations while the user machine is otherwise idle. Once the blocks are processed, results are returned to the central server. This way, users are able to donate their spare cycles to a cause that many people regard as legitimate.

In a different community, we have witnessed the recent emergence of a class of peer-to-peer applications that have achieved widespread deployment among common Internet users (e.g., BitTorrent’s peer-to-peer content distribution network represented 35% of Internet’s traffic two years ago [1]). Relevant research work in the area of peer-to-peer networks, namely to build robust distributed hash-tables, includes projects such as Chord [17], and Pastry [16].

**Shortcomings of Current Solutions:** Despite its potential and success stories, the Grid revolution has failed to reach the common Internet user. A user outside the domain of the corporate or scientific communities, that currently use the Grid, has to overcome several barriers before it can deploy its own application on the Grid.

For that matter, the user has to either negotiate the access to existing Grid infrastructures, or set up its own infrastructure. The former option is perceived as difficult, and, if the managers of existing Grid infrastructures were to allow unlimited access to their computing resources, this might easily lead to the collapse of that infrastructure. The latter option may not be feasible for every user if he does not have access to the needed resources.

In the case of popular distributed cycle sharing efforts such as SETI@home, people are fundamentally motivated to contribute to a noble collective cause, and with equal concern, in improving their ranking within the users community [4]. Nevertheless, in the kind of peer-to-peer infrastructure we project, home users can also help themselves, while helping others do the same. In some sense, the desiderata of selflessness and altruism are replaced by actual mutualism, since users must be allowed to execute applications for themselves, instead of just for collective causes.

Decentralized P2P research projects and file-sharing applications, although successful in providing high availability and attracting users (and public controversy), are very limited because they do not leverage the computing

power available in participating nodes.

Peer-to-peer architectures and grid computing have been combined in several projects (e.g., Integrate [11], OurGrid [5]). However, they only aim at dynamically federating (e.g., for scheduling) grid infrastructures (e.g., clusters) that are already deployed, and therefore share the same drawbacks of institutional grids.

The combination of decentralized peer-to-peer architectures and distributed cycle sharing has also been addressed, as in the case of CCOF [13]. However, such an infrastructure was never successfully deployed by vast numbers of home users. We argue that part of this was due to the inability to provide home users with a variety of interesting use cases.

**Contribution and Roadmap:** Our goal is to achieve a synthesis of the three approaches (institutional grid infrastructures, distributed cycle-sharing, and decentralized P2P architectures) by building the substrate for a peer-to-peer Grid infrastructure called GINGER (Grid Infrastructure for Non-Grid Environments), or simply **GiGi**, that can enable the widespread use of Grid technologies by home users.

To address the issue of providing users with a variety of interesting applications that can be run on this infrastructure, we present a novel application and programming model that revolves around the concept of a *Gridlet*, a semantics-aware unit of workload division and computation off-load. Moreover, the concept of gridlet can also be employed to drive resource management, and accounting of peer contribution.

A gridlet is a chunk of data associated with the operations to be performed on the data. We envision that, in most cases, such operations can consist of unmodified application binaries. This will enable a trivial adoption of this model, and provide improved performance of popular applications executed regularly by many users, if they are allowed to use idle computing power in other nodes.

Examples of popular applications that can use this model are audio and video compression, signal processing related to multimedia content (e.g., photo, video and audio enhancement, motion tracking), content adaptation (e.g., transcoding), and intensive calculus for content generation (e.g., ray-tracing, fractal generation).

GiGi brings the Grid to home-users, by transforming the present examples of "grid-that-matters" (e.g., for climate prediction), into a "grid-for-the-masses" (e.g., to achieve faster video compression).

The rest of this paper is organized as follows. In the next section we provide a system overview of GiGi. Section 3 is dedicated to the Gridlet application and programming model. Section 4 addresses resource management, i.e., namely gridlet accounting. Section 5 covers the main implementation issues. We discuss relevant related work in Section 6, and finish the paper with some conclusions and future work.

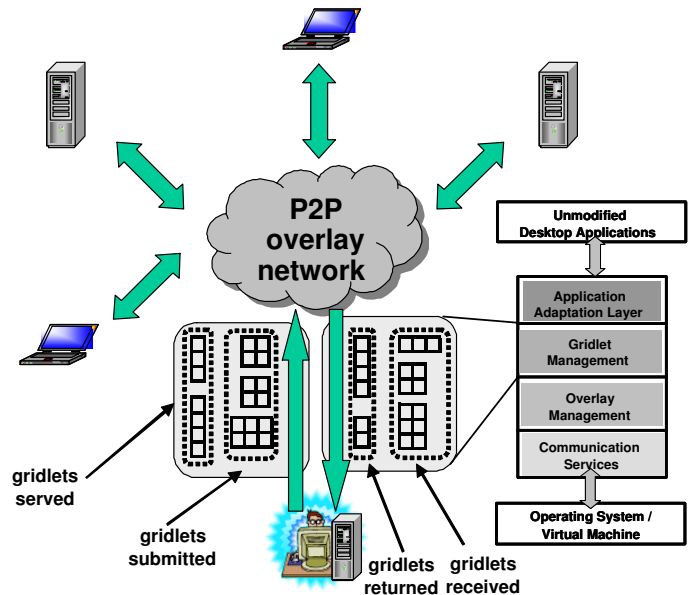


Figure 1. GiGi: An Ocean of Gridlets

## 2 System Overview: Ocean of Gridlets

Figure 1 depicts a global view of the GiGi architecture. The basic unit of work which may be deployed in GiGi is called a Gridlet. It is semantically-enriched to fully describe a data workload as well as the transformations required on it.

GiGi provides an abstraction of an *Ocean of Gridlets* that are portrayed as sets of small squares. Gridlets are *submitted* by nodes and flow across the peer-to-peer overlay to be *received* and *served* by other nodes, and later *returned* as gridlet-results to the submitting nodes.

Gridlet data itself is mostly opaque to GiGi. A gridlet may or may not carry the actual code that will process the gridlet's data. Gridlets must have an estimate of cost associated with them, depicted as gridlets of different size. This cost is represented as vector  $\vec{C} = (\text{CPU}, \text{BW})$ , representing both CPU and bandwidth costs of transferring and processing a gridlet.

**GiGi Middleware:** The GiGi middleware runs on each node enrolled in a GiGi grid-overlay, and follows a layered architecture to favor portability and extensibility (also depicted in Figure 1). The GiGi middleware, via the Overlay Management layer, is responsible for maintaining the overlay network to exchange gridlets with other nodes. Actual network transfer is carried out by the Communication Services.

The Gridlet Management layer performs the tasks necessary to partition files into properly formed gridlets, and later reassemble gridlet results, in order to generate result files. The Application Adaptation layer is responsible for interacting with the actual unmodified desktop applications, e.g., launch them, feeding the data inside gridlets, and collecting results.

**GiGi Overlay Management:** In GiGi, nodes are organized in a peer-to-peer overlay such as Pastry [16]. This

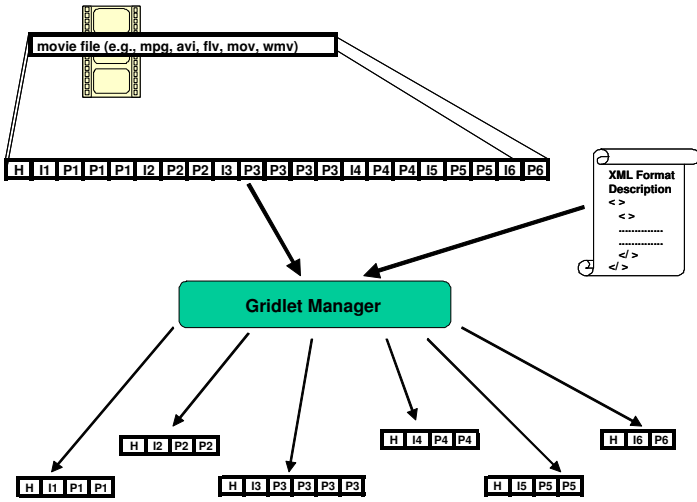


Figure 2. Gridlet Creation for a movie file

allows resource discovery protocols to follow overlay links, and inherit several of the good properties of the overlay, such as adjusting automatically to frequent membership changes.

To reduce bandwidth and CPU costs, we intend to heavily exploit the caching of code, gridlet input, and gridlet results, since gridlets are regarded as deterministic. If two users run the same computation (e.g., converting the same movie to the same format) by submitting a number of gridlets to the overlay, each one should profit from the other’s gridlet-processing, and download gridlet-results from it right away.

In general, any submitting node should get gridlet-results from those nodes that already contributed to performing the same computation, earlier, over the same data. For that matter, a peer-to-peer DHT is well suited for storing a small index of where the owners of cached results can be located. Furthermore, gridlet execution may be combined with replication for higher availability and to allow screening from forged results.

### 3 Gridlet Application Model

In this section, we describe the Gridlet Application Model, resorting to a prototypical example of a video transcoding utility, as depicted in Figures 2 and 3.

The gridlet application model divides application execution in the following phases: 1) gridlet creation, 2) gridlet processing, and 3) gridlet-result aggregation. Gridlet processing can be further divided in: 2a) gridlet-data injection, 2b) application execution, and 2c) gridlet-result extraction. A gridlet is always in one of the states  $\{REQUEST, EXECUTION, RESULT, ERROR\}$ . A gridlet is in REQUEST state when it is created submitted to the Gigi overlay. When some peer accepts to service a gridlet, it transits to the EXECUTION state. When processing is complete, the gridlet transits to the RESULT state and becomes available to be returned. Gridlets finished with incomplete result data or causing errors, are in ERROR state, and may be silently discarded.

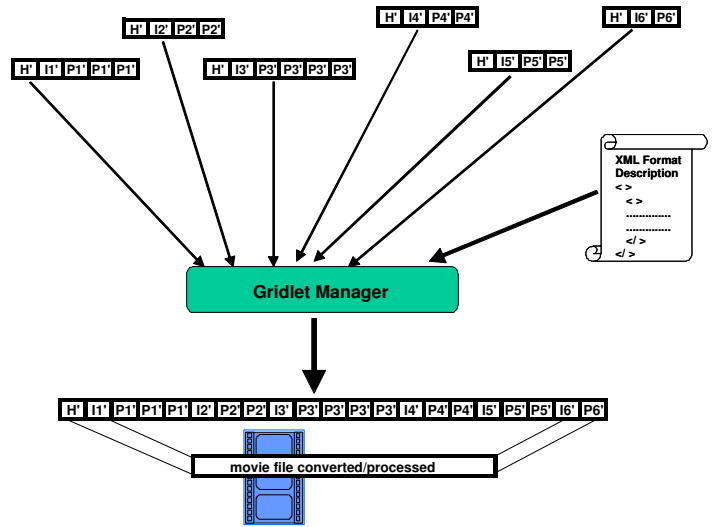


Figure 3. Reassembly of Gridlet-Results

**Gridlet Creation:** The first stage of gridlet creation is data-partitioning. In its most basic form, data-partitioning may be performed simply by partitioning the file(s) holding the data to be processed. However, this may not be as simple, as we describe next.

The Gridlet Manager is the GiGi middleware component that is responsible for semantics-aware (no-nonsense) data-partitioning, and appending required prefixes and/or suffixes to gridlet data, so that it may be processed transparently by non-gridlet-aware code of unmodified desktop applications. The Gridlet Manager is driven by XML-based format descriptions. These depict file formats, namely regarding partitioning and reassembly points, headers that must be adapted and included in each gridlet, and fragments of files that should be kept within the same gridlet.

As an example, consider the unmodified video transcoding utility that is to be executed in GiGi, depicted in Figure 2. Splitting the movie file in several fragments, place each of them in a separate gridlet, and execute the video transcoding application on each of them, will not produce the desired results. The transcoding utility is unable to decode arbitrary fragments of a video file. It must be fed with complete frames (or pictures). More so, it needs to access the movie header information to be able to decode it, and each *predicted* picture, containing only deltas, must be in the same gridlet that contains the full (*intra*) picture it is based on. The Gridlet Manager, consulting the XML format description, is able to correctly partition the file data, in individual frames. Then, it creates gridlets of different sizes suitable to the intended application ( $\{H, I1, P1, P1\}$  through  $\{H, I6, P6\}$ ). They all contain a proper file header (as if they were a short movie), and all *predicted* pictures are placed in the same gridlet of the picture they are based on (e.g., P6 and I6 frames are placed in the same gridlet).

**Gridlet Processing:** Data inside gridlets may be provided to execution in several ways. The most fundamen-

tal form of gridlet data-injection (widely used in Grid infrastructures and desktop applications) is file semantics and runtime arguments. Gridlet code being executed is allowed to randomly access gridlet data for its input as if it was a file. Alternatively, gridlet-data could be injected into application execution via a pipe, or as a high-level language data-structure (e.g., a Java array for a Java-based gridlet).

Peers servicing gridlet execution need not trust gridlet code. The GiGi runtime does not execute gridlets directly. Gridlets that are bytecode-based are executed in the context of a sandbox provided by its virtual machine. Gridlets based on native code should be executed in the context of a general-purpose virtual machine, such as VMWare’s, VirtualServer, or Xen. Naturally, gridlet code may be subject to regular certification procedures in order to increase peer confidence. Hopefully, for popular applications, the application code will already reside at the servicing peer (it may also service its own gridlets).

Once gridlet execution is complete, the results of the application execution must be extracted to update the gridlet, now in the RESULT state. i.e., a gridlet-result.

In the example described in this section, gridlet-data injection and result extraction are performed via the input and output files of the video transcoding utility. Gridlet-results are updated versions of the gridlet-request received ( $\{H',I1',P1',P1'\}$  through  $\{H',I6',P6'\}$ ).

In general, the data payload of a gridlet may vary from REQUEST to RESULT state. For some applications gridlet-results may have but a fraction of data size of the corresponding gridlet-request (e.g., cryptographic challenge), while others may have substantially more (e.g., high-resolution ray-tracing).

**Gridlet-Result Reassembly:** Once the processing of a gridlet is completed, the gridlet-result becomes available for being returned back to the submitting peer. The final part of the execution of a gridlet-based application is result gathering and reassembly, portrayed in Figure 3.

Gridlet-results must be fetched back from whatever nodes have serviced them and reassembled (i.e., recombined) in a result file. In the example, the resulting transcoded video file must have one (and only one) proper header, and all its frames correctly ordered. The Gridlet Manager, driven by the XML format description, is able to reassemble all gridlet-results, according to application and format semantics, into a playable result movie file.

Some application might not be subject to ordering constraints, or not even need to all gridlet-results. A cryptographic challenge does not need any of its results, except the one that defeats the proposed challenge.

**Gridlet Representation:** Gridlets are represented internally as XML documents comprised of two headers and payload. One header carries information to be used by the Gridlet Manager (e.g., gridlet-cost and sequencing information). The other is to be used by the Application Adaptation Layer (e.g., application names or GUIDs, URLs, etc., and parameter-passing).

Gridlets should not be large to be easily routed around nodes and serviced. A gridlet may be regarded as an enhancement of a file fragment (in BtTorrent) or part (in eMule), that will be processed by an unmodified application instead of just stored and exchanged. Therefore, common sizes for these fragments or blocks can also be employed for gridlets (e.g., 64 KB, 256 KB).

## 4 Resource Management

Gridlets do not have all the same associated cost, both in terms of CPU and bandwidth cost. Some may require more data to be carried, while others may be more computational intensive. Therefore, we require a standard unit of measurement for gridlet cost.

Resource currency in GiGi is the unit vector  $\vec{G}\$=(1,1)$ , w.r.t. both CPU and bandwidth. The standard unit, itself, is solely instrumental, and does not have to carry special meaning. Naturally, it is practical if  $\vec{G}\$$  is associated with the total cost of servicing a *standard* gridlet in a *standard* machine.

Gridlet costs are always represented against the standard-gridlet, e.g.,  $\vec{G}\$(5.5, 2.75)$ . The  $\vec{G}\$$  currency unifies resource management in GiGi. It is used to describe gridlet cost, and is the basis upon which resource contribution and availability are measured. Gridlet costs may be added simply by using regular vector addition.

**Gridlet Cost Estimation:** Ultimately, the cost of a gridlet would equate to a number of CPU-cycles and bandwidth required. Therefore we can define a  $\vec{G}\$$  standard unit arbitrarily, e.g., of (1 MFLOP, 1 KB) or a much larger one of (1 GFLOP, 1 MB). An alternative and practical approach is to define  $\vec{G}\$$  as the cost of solving the Linpack [8] benchmark, paired with the bandwidth required to transfer the benchmark test-data used.

Since the cost of a given gridlet is always assessed against the cost of servicing the standard-gridlet, gridlet cost estimation can be easily achieved. The gridlet submitting peer needs only process some sample gridlets locally, and weight the cost of their execution against current CPU-load, and against the average time to compute the standard-gridlet in its computer.

Nonetheless, it is only natural that the perceived computational cost of a gridlet will actually vary from system to system. Nonetheless, as gridlets are processed, the servicing peer is able to produce its own estimate of their computational cost, and use it for future accounting w.r.t. similar gridlets.

If there are relevant variations between submitted and locally calculated gridlet costs, the servicing peer may include this notification in the gridlet-result informing the submitter and other peers. Conversely, when gridlet-results are returned, submitting peers may double-check cost-claims of the servicing peers.

**Accounting Peer Availability and Contribution:** Gridlet costs may be subject to management activities such as integration of gridlet cost serviced over a specific time-interval. Likewise, it is possible to estimate peer

capabilities as a function of the availability to serve a number of gridlets during a unitary time interval (one second, minute, hour), i.e., gridlet processing *power*, or over a period of time, i.e., gridlet processing *work* (as KW and KWh in electric grids, respectively).

Thus, peer contribution to GiGi is evaluated by statistical functions over gridlet availability, i.e., gridlet requests submitted, received, and gridlet-results returned, during a specific period of time, both in number as well as their aggregate cost. Conversely, peer usage of GiGi is evaluated by gridlet requests submitted, and gridlet results received, during a specific period of time.

Since peers are expected to connect to GiGi using their personal computers, they should be able to define their contribution to GiGi in order not to disturb their regular usage. A special case of this is to execute the GiGi runtime as a low-priority process, limiting the contribution to the intended  $G^{\$}$  value over time. This technique has been adopted in earlier distributed computing voluntary efforts [4] with great success. We believe that near cost-free participation is quintessential to GiGi adherence and proliferation.

As with every parallel distributed application, the trade-off between remote execution of code and data transfers versus local execution must be addressed. The use of GiGi, gridlets, and run-time data partitioning eases the evaluation of the best remote execution strategy, e.g., users can interactively instruct the gridlet managers to create gridlets with maximum execution cost while reducing bandwidth cost.

## 5 Implementation Issues

We are currently implementing the gridlet application model using a Java prototype of GiGi. It performs Gridlet Management, and Application Adaptation is limited to launching applications whose input and output files can be provided as runtime arguments. Gridlet exchanging among nodes employs an overlay set-up with JXTA. The use of Java does not preclude the execution of unmodified applications. Java code simply creates and reassembles gridlets, driven by XML format descriptions, and invokes desktop applications as native-code processes.

Following, we will address the integration of the gridlet model with open-source versions of popular P2P file-sharing tools, such as BitTorrent or eMule. These will be used for overlay management, gridlet compression, and user interaction, reusing the gridlet management code.

## 6 Related Work

The most significant implementation of a standards-based Grid infrastructure is the Globus toolkit [9], that implements services for user authentication and authorization, job submission and data transfer, among others. The programming model in Globus is based on the use of specialized libraries for remote job submission and control like GRAM.

In Condor [19], another important infrastructure for resource sharing in the context of High-Throughput Com-

puting (HTC), users can submit executable binaries that are run on remote machines, and can read/write their input/output from/to a shared file system. They also provide separate support for file transfer, when a shared file system is not available. All of these features (like which files need to be copied and to where) have to be configured by the user that wants to run a remote job, which contrasts with our vision of users being given a generic tool that they can run and will perform all the necessary work seamlessly.

Cluster Computing on the Fly (CCOF) [13] is a project that shares our goals of deploying a generic peer-to-peer grid infrastructure, although with many relevant design differences. However, they have focused on issues like resource discovery, and detection of incorrect results, and to the best of our knowledge have not detailed the programming model.

InteGrade [11] is a middleware that constructs a hierarchy of clusters to speed-up MPI-based applications, over a lightweight CORBA implementation. Resource management is based on usage pattern analysis that monitors available resources over periods of time, and determines relevant situation categories (e.g., lunch, weekend, etc.). Though effective in harnessing spare cycles from other nodes, it is designed for a traditional parallel programming model, which is not adopted by most popular desktop applications.

The OurGrid [5] project also federates sites with grid clusters in a peer-to-peer manner, by scheduling applications to run on remote nodes, and managing resources using a Network of Favours. It is also unable to exploit the parallelism, at the data-level, of popular desktop applications, as the gridlet model does.

The work in P3 [14] proposes a parallel programming environment based on an underlying peer-to-peer infrastructure. Participating machines may behave as computing nodes, that perform actual computations, or manager nodes, that behave as Gnutella [15] ultra-peers. Applications are exclusively developed in Java and must derive from a specific class (P3Parallel), which precludes our vision of fostering adoption of the system by running existing applications unmodified. P3 also contrasts in the fact that it tries to build a peer-to-peer storage system to save intermediate results, and also for process synchronization and message passing. This raises bandwidth issues w.r.t. maintaining data available in the presence of a dynamic membership [7]. Once again, an approach based on stateless components, like we prescribe, is more adequate to a peer-to-peer deployment.

Triana [18] provides an alternative programming model for Grid applications that is based on graphical component composition for task-graphs. Although it uses peer-to-peer technologies, their programming model is quite different from ours, since it requires using a specific tool to develop applications.

Xtremweb [10] proposes a three-tier model for a peer-to-peer based parallel programming system. Clients submit jobs to a central coordinator that manages a commu-

nity of users. Worker nodes, as they become available, pull work jobs from the coordinator. The presence of such a central component is probably not very adequate to the environments we are addressing, where the failures are the norm and not the exception. Unlike our proposals, this system uses a traditional parallel programming model based on RPC and MPI is not the most appropriate for the scenarios we envision.

BOINC [3] is a distributed computing platform developed at Berkeley. It has surpassed its original project, SETI@home, and encompasses now a large number of related projects. BOINC contains the notion of work unit but it is not flexible. Every work unit is regarded as having the same computational and bandwidth cost, determined by each project. It is impossible to know, in advance, e.g., whether it costs more CPU or bandwidth to serve a block of SETI@home or of Folding@home. Furthermore, users cannot submit their own work units without having to develop a full-fledged BOINC client and setting up their BOINC server. This is a much more inflexible and complex approach than one based on the gridlet application model.

Mosix [6] is a management system for Linux-based systems. It is able to combine several nodes into a Linux cluster, offering transparent process migration, and system-call redirection. However, it does not exploit data-parallelism automatically as the gridlet model. Also, users probably will not replace their operating system.

In summary, although there is relevant related work and successful projects in the areas of grid computing, distributed cycle-sharing, and peer-to-peer computing, to the best of our knowledge, none of them provides an application model that offers improved performance, with transparency, to existing applications executed by Internet home users.

## 7 Conclusion

In this paper, we presented a new application model based on the concept of *gridlet* that can bridge the gaps between a number of existing infrastructures (i.e., grids, distributed cycle-sharing, and decentralized P2P file-sharing), bringing Grid technology to home users.

We described how the gridlet model is employed to adapt application execution in the context of a generic peer-to-peer grid infrastructure: Gigi. We also described how gridlets are used to estimate computation cost, and to manage resources and peer-contribution.

Contrary to previous approaches, it is able to transparently enhance the performance, by using cycles of idle nodes, of unmodified popular desktop applications usually executed. This key novel feature overcomes the limitations w.r.t. worldwide deployment of previous work.

In the future, we plan to further develop the GiGi prototype and experiment with adaptation to several applications and formats, and other overlay configurations. We intend to mechanically derive XML descriptions for the formats handled by each specific application, based on current parsers for those formats.

## References

- [1] Bittorrent accounts for 35% of internet traffic, Slashdot article referring to an internet traffic study, nov 2004.
- [2] M. Allen. Do-it-yourself climate prediction. *Nature*, 401(6754):642–642, 1999.
- [3] D. P. Anderson. Boinc: a system for public-resource computing and storage. In *Proceedings. Fifth IEEE/ACM International Workshop on Grid Computing*, 2004.
- [4] D.P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. SETI@ home: an experiment in public-resource computing. *Communications of the ACM*, 45(11):56–61, 2002, data regarding fake results in <http://www.openp2p.com/pub/a/p2p/2001/02/15/anderson.html>.
- [5] N. Andrade, L. Costa, G. Germoglio, and W. Cirne. Peer-to-peer grid computing with the ourgrid community. In *23rd Brazilian Symposium on Computer Networks (IV Special Tools Session)*, May 2005.
- [6] Amnon Barak, Amnon Shiloh, and Lior Amar. An organizational grid of federated mosix clusters. In *Proc. IEEE International Symposium on Cluster Computing and the Grid*, 2005.
- [7] Charles Blake and Rodrigo Rodrigues. High availability, scalable storage, dynamic peer networks: Pick two. In *Ninth Workshop on Hot Topics in Operating Systems (HotOS-IX)*, pages 1–6, Lihue, Hawaii, May 2003.
- [8] J.J. Dongarra. Performance of various computers using standard linear equations software. *ACM SIGARCH Computer Architecture News*, 20(3):22–44, 1992.
- [9] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. In *Proc. of the Workshop on Environments and Tools for Parallel Scientific Computing, SIAM*, August 1996.
- [10] Cecile Germain, Vincent Nori, Gilles Fedak, and Franck Cappello. Xtremweb: Building an experimental platform for global computing. In *GRID '00: Proceedings of the First IEEE/ACM International Workshop on Grid Computing*, pages 91–101, London, UK, 2000. Springer-Verlag.
- [11] A. Goldchleger, F. Kon, A. Goldman, M. Finger, and G.C. Bezerra. InteGrade: object-oriented Grid middleware leveraging the idle computing power of desktop machines. *Concurrency and Computation: Practice & Experience*, 16(5):449–459, 2004.
- [12] S.M. Larson, C.D. Snow, M. Shirts, and V.S. Pande. Folding@ home and genome@ home: Using distributed computing to tackle previously intractable problems in computational biology. *Computational Genomics*, 2002.
- [13] Virginia Lo, Daniel Zappala, Dayi Zhou, Yuhong Liu, and Shanyu Zhao. Cluster computing on the fly: P2p scheduling of idle cycles in the internet. In *3rd International Workshop on Peer-to-Peer Systems (IPTPS 2004)*, 2004.
- [14] Licinio Oliveira, Luis Lopes, and Fernando M. A. Silva. P3: Parallel peer to peer. In *Revised Papers from the NETWORKING 2002 Workshops on Web Engineering and Peer-to-Peer Computing*, pages 274–288, London, UK, 2002. Springer-Verlag.
- [15] M. Ripeanu. Peer-to-Peer Architecture Case Study: Gnutella Network. *Proceedings of International Conference on Peer-to-peer Computing*, 101, 2001.
- [16] Antony Rowstron and Peter Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, November 2001.
- [17] I. Stoica, R. Morris, D. Karger, M.F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *Proceedings of the 2001 SIGCOMM conference*, 31(4):149–160, 2001.
- [18] Ian Taylor, Matthew Shields, Ian Wang, and Omer Rana. Triana Applications within Grid Computing and Peer to Peer Environments. *Journal of Grid Computing*, 1(2):199–217, 2003.
- [19] Douglas Thain, Todd Tannenbaum, and Miron Livny. Condor and the grid. In Fran Berman, Geoffrey Fox, and Tony Hey, editors, *Grid Computing: Making the Global Infrastructure a Reality*. John Wiley & Sons Inc., December 2002.