Grid File Transfer during Deployment, Execution, and Retrieval

Françoise Baude, Denis Caromel, Mario Leyton, and Romain Quilici

INRIA Sophia-Antipolis, CNRS, I3S, UNSA. 2004, Route des Lucioles, BP 93, F-06902 Sophia-Antipolis Cedex, France.
First.Last@sophia.inria.fr

Abstract. We propose a file transfer approach for the Grid. We have identified that file transfer in the Grid can take place at three different stages: deployment, user application execution, and retrieval (post-execution). Each stage has different environmental requirements, and therefore we apply different techniques. Our contribution comes from: (*i*) integrating heterogeneous Grid resource acquisition protocols and file transfer protocols including deployment and retrieval, and (*ii*) providing an asynchronous file transfer mechanism based on active objects, wait-by-necessity, and automatic continuation.
We validate and benchmark the proposed file transfer model using ProActive, a Grid programming middleware. ProActive provides, among others, a Grid infrastructure abstraction using deployment descriptors, and an active object model using transparent futures.

1 Introduction

Scientific and engineering applications that require, handle, and generate large amount of data represent an increasing use of Grid computing. To handle this large amount of information, file transfer operations have a significant importance. For example, some of the areas that require handling large amount of data in the Grid are: bioinformatics, high-energy physics, astronomy, etc.

Although file transfer utilities are well established, when dealing with the Grid, environmental conditions require reviewing our previous understanding of file transfer to fit new constraints and provide new features at three different stages of Grid usage: deployment, execution, and post-execution. At deployment time, we focus on integrating heterogeneous file transfer and resource acquisition protocols to allow *on-the-fly* deployment. During the application run time, we offer a parallel and asynchronous file transfer mechanism based on active objects, wait-by-necessity, and automatic continuation. Once the user application has finished executing, we offer a file retrieval mechanism.

This document is organized as follows. In section 2 we provide some background on the Grid programming middleware ProActive. In sections 3 and 4 we describe our file transfer proposal for the Grid, and show how this is implemented in the context of ProActive. We benchmark the implementation of the model in section 5. Related work is reviewed in section 6, and finally we conclude in section 7.

2 Background on ProActive

Figure 1, shows the *active object* (AO) programming model used in ProActive[16]. AO are remotely accessible via method invocations, automatically stored in a queue of pending requests. Each AO has its own thread of control and is granted the ability to decide in which order incoming method calls are served (FIFO by default). Method calls on AO are asynchronous with automatic synchronization (including a rendezvous). This is achieved using automatic *future objects* as a result of remote methods calls, and synchronization is handled by a mechanism known as *wait-by-necessity* [4].

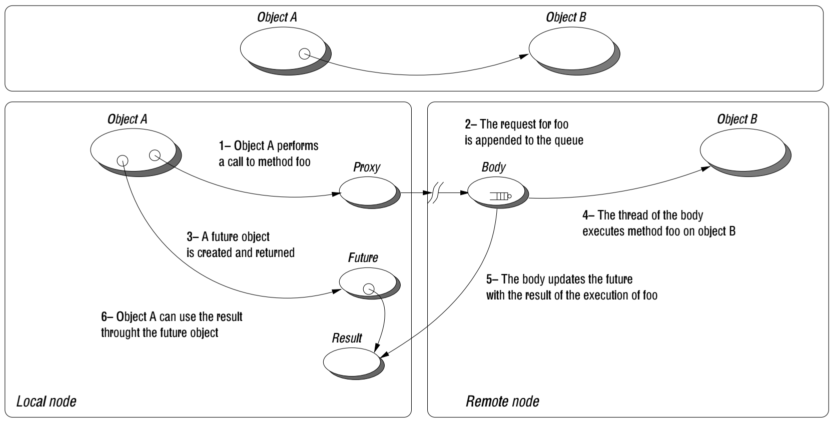

Object A
Object B
Object A
1– Object A performs a call to method foo
Proxy
2– The request for foo is appended to the queue
Body
Object B
4– The thread of the body executes method foo on object B
3– A future object is created and returned
Future
5– The body updates the future with the result of the execution of foo
6– Object A can use the result throught the future object
Result
Local node
Remote node

Fig. 1. Execution of a remote method call.

ProActive also provides a *Descriptor Deployment Model* [3], which allows the deployment of applications on sites using heterogeneous protocols, without changing the application source code. All information related with the deployment of the application is described in the XML Deployment Descriptor. Thus, eliminating references inside the code to: machine names, resource acquisition protocols (local, rsh, ssh, lsf, globus-gram, unicore, pbs, lsf, nordugrid-arc, etc..) and communication protocols (rmi, jini, http, etc...).

The Descriptor Deployment Model is shown in Figure 2. The infrastructure section contains the information necessary for booking remote resources. Once booked, ProActive Nodes can be created (or acquired) on the resources. To link the Nodes with the application code, a Virtual Node (VN) abstractions is provided, which corresponds to the actual references in the application code. Virtual Nodes have a unique identifier which is hardcoded inside the application and the descriptor.

A deployer can change the mapping of the application → Virtual Node to deploy on a different Grid, without modifying a single line of code in the application.

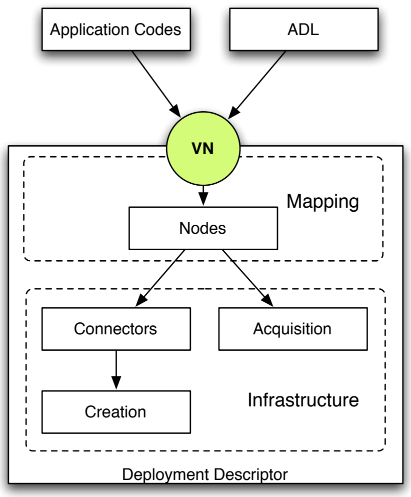


Fig. 2. Descriptor Deployment Model

3 Grid Deployment and File Transfer

3.1 On-the-fly Deployment

We consider that deployment on the Grid represents the fulfillment of the following tasks: (*i*) Grid infrastructure setup (protocol configuration, installation of Grid middleware libraries, etc...), (*ii*) resource acquisition (job submission), (*iii*) application specific setup (installing application code, input files, etc...), and (*iv*) application deployment (setting up the logic of the application).

Usually, the deployment requires files transfer during the above cited tasks to succeed, for such files as: Grid middleware libraries (*i*), application code (*iii*), and application input files (*iv*). We say a Grid deployment can be achieved **on-the-fly** if the required files can be transferred when deploying, without having to install them *in advance*. It is our belief, that on-the-fly deployment greatly reduces the Grid infrastructure configuration, maintenance and usage effort.

In the rest of this section, we describe how heterogeneous protocols for file transfer and resource acquisition can be integrated to achieve on-the-fly deployment for the Grid. To explain the approach, we first introduce the notation and review some general concepts concerning resource acquisition and file transfer.

3.2 Concepts

Let r be a resource acquisition protocol, t a file transfer protocol, n a Grid node, p a Grid infrastructure parameter, and f a file definition. We say a node n_k is acquirable from n_0 iff $\exists\{r_0(p_0), \ldots, r_{k-1}(p_{k-1})\}$ and $\exists\{n_0, \ldots, n_{k-1}\}$ as shown in Figure 3(a). The nodes are acquired sequentially one after the other, i.e. n_k is acquired before n_{k+1} using a resource acquisition protocol r_k.

A Grid infrastructure resource acquisition can more precisely be seen as a tree, since more than one node can be acquired in parallel. As shown in Figure 3(b), the leaf nodes represent the acquired resources[1], and we will call them `virtualNode`, using the ProActive terminology.

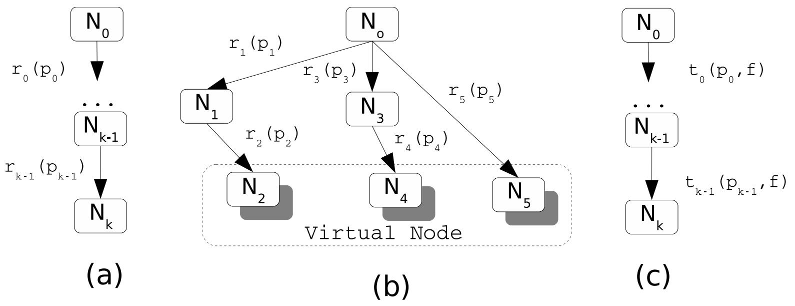


Fig. 3. Resource Acquisition and File Transfer.

Given a file transfer protocol t we say a file f can be transferred from n_0 to n_k iff $\exists\{t_0(p_0, f), \ldots, t_{k-1}(p_{k-1}, f)\}$ and $\exists\{n_0, \ldots, n_{k-1}\}$ (Figure 3(c)).

A file transfer protocol can be of two types: internal if the file transfer protocol is executed by the resource acquisition protocol, i.e. $r(p, f)$ executes the file transfer and performs the resource acquisition (`unicore`, `nordugrid`); or external if they are not part of a resource acquisition protocol (`scp`, `rcp`). Therefore, internal file transfer protocols can not be used separately from the corresponding resource acquisition protocol.

3.3 Integration Proposal

Supposing that n_{k+1} is acquirable from n_k using r_k, and given an ordered list of file transfer protocols $\overrightarrow{t_k}$ that can or cannot be successful at transferring f from n_k to n_{k+1}. Then, if there $\exists t_k^i \in \overrightarrow{t_k}$ which corresponds to the lower indexed transfer protocol capable of transferring f, we propose the sequencing of file transfer and resource acquisition protocols in the following way:

1. If t_k^i is external, then we will execute

$$n_k \xrightarrow{t_k^0(p,f),\ldots,t_k^i(p,f),r_k(p)} n_{k+1}$$

[1] Depending on the deployment mechanism, sometimes the internal nodes also represent acquired resources.

That is to say, that the file transfer protocols will be executed sequentially until one of them succeeds, and then the resource acquisition protocol will be executed.

2. If t_k^i is an internal file transfer protocol of r_k, then we will execute:

$$n_k \xrightarrow{t_k^0(p,f),\ldots,t_k^{i-1}(p,f),r_k(p,f)} n_{k+1}$$

The assumption is that the internal t_k^i of a given r_k will always succeed. This is reasonable, because if the internal t_k^i fails, this implies that r_k will also fail, and thus there is no point on testing further file transfer protocols.

The problem with the sequencing approach, is that no file transfer protocol $t_k^i \in t_k$ may be successful at transferring f. To solve this, we propose the usage of a *failsafe* file transfer protocol, which is reliable at performing the file transfer, but only after the resource acquisition has taken place. Therefore, if t_k^i is a failsafe protocol, then we will execute:

$$n_k \xrightarrow{t_k^0(p,f),\ldots,t_k^{i-1}(p,f),r_k(p),t_k^i(p,f)} n_{k+1}$$

Note that in the failsafe approach, the actual file transfer is performed after the resource acquisition.

There are two main reasons for trying to avoid using a failsafe protocol. The first one, is that failsafe performs the file transfer at a higher level of abstraction, not taking advantage of lower level infrastructure information, as shown in the benchmarks of section 5.2. The second reason is that *on-the-fly* deployment becomes limited: the libraries required to use the failsafe protocol cannot be transferred using the failsafe protocol, and must be transferred in advance.

3.4 File Transfer in ProActive Deployment Descriptors

Figure 4 shows how the approach is integrated into ProActive XML Deployment Descriptors. We take advantage of the descriptors structure to apply separation of concerns. The actual files requiring file transfer are specified in a different section (`FileTransferDefinitions`) than the Grid infrastructure parameters (`FileTransferDeploy`). The infrastructure parameters holds information such as: the sequence of protocols that will be tried to copy the file (`copyProtocol`)[2], hostnames, usernames, etc. Finally, the `FileTransferRetrieve` tag specifies which files should be retrieved from the nodes in the retrieval (post-execution) phase (reviewed in further depth in section 4.2).

4 File Transfer during execution and retrieval

Applications can generate data, and transferring this data during the application execution is usually achieved using a specific communication protocol for

[2] The *failsafe* protocol shown in the example is described in further detail in section 4.1.

```
<FileTransferDefinitions>
   <FileTransfer id="requiredfiles">
      <file src="application.class" dest="application.class"/>
      <file src="ProActive.jar" dest="ProActive.jar"/>
      <file src="input.dat" dest="input.dat"/>
  </FileTransfer>
  <FileTransfer id="results"><file src="output.dat"/></FileTransfer>
<FileTransferDefinitions>
...
<virtualNode name="exampleVNode" FileTransferDeploy="requiredfiles"/>
...
<processDefinition id="xyz">
  <sshProcess>
    <!-- The refid attribute can be set to "implicit", which will use the value defined in
         the VirtualNode. -->
    <FileTransferDeploy refid="implicit">
      <copyProtocol>processDefault, rcp, scp, failsafe</copyProtocol>
      <sourceInfo prefix="/home/user"/>
      <destinationInfo prefix="/tmp" hostname="foo.org" username="smith" />
    </FileTransferDeploy>
    <!-- The refid can also directly reference the FileTransfer id. -->
    <FileTransferRetrieve refid="results">
      <sourceInfo prefix="/tmp"/>
      <destinationInfo prefix="/home/user"/>
    </FileTransferRetrieve>
  </sshProcess>
</processDefinition>
```

Fig. 4. Example of File Transfer in Deployment Descriptor.

transferring the file's data. Nevertheless, Grid resources are characterized by distributed ownership and therefore diverse management policies, as our own experiments [14, 15] confirm it. As a result, setting up the Grid to allow message passing is a painfull task. Additionally configuring and maintaining a specific file transfer protocol between any pair of nodes seems to us as an undesirable burden[3].

Therefore, we propose that the file transfer protocol should be built on top of other protocols, specifically the message passing protocols. Standard message passing is not well suited for transferring large amounts of information, mainly because of memory limitations and lack of performance optimizations for large amounts of data. In this section we show how an active object based message passing model can be used as the ground for a portable efficient scalable file transfer service for large files, where large means bigger than available runtime memory. Additionally by using active objects as transport layer for file transfer, we can benefit from the automatic continuation to improve the file transfer between peers, as we will show in the benchmarks of section 5.

4.1 Asynchronous File Transfer with Futures

We have implemented file transfer as service methods available in the ProActive library as shown in Figure 5. Given a ProActive **Node** *node*, a **File**(s) called

[3] Deployment file transfer does not impose this burden, because the file transfer does not take place between each possible pair of nodes.

source, and a **File**(s) called *destination*, the *source* can be pushed (*sent*) or pulled (*get*) from *node* using the API. The figure also shows a `retrieveFiles` method, which is discussed in section 4.2.

```java
//Send file(s) to Node node
static public File pushFile(Node node, File source, File destination);
static public File[] pushFile(Node node, File[] source, File[] destination);

//Get file(s) from Node node
static public File pullFile(Node node, File source, File destination);
static public File[] pullFile(Node node, File[] source, File[] destination);

//Retrieve files specified for the virtualNode
public File[] virtualNode.retrieveFiles();
```

Fig. 5. File Transfer API.

The *failsafe* algorithm mentioned in section 3.3 is implemented using the `pushFile` API, which is itself built using the *push* algorithm depicted in Figure 6 and detailled as follows:

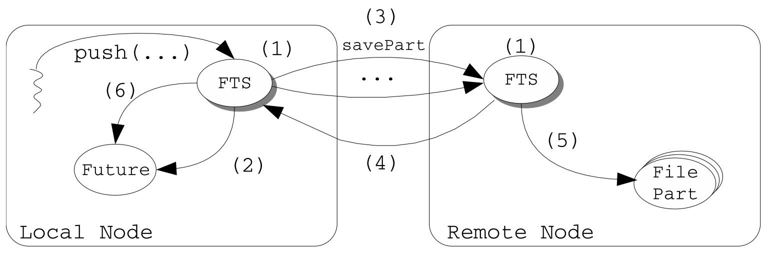


Fig. 6. Push Algorithm.

1. Two File Transfer Service (FTS) active objects are created (or obtained from a pool): a local FTS, and a remote FTS. The *push* function is invoked by the caller on the local FTS: $LocalFTS.push(\ldots)$.
2. The local FTS immediately returns a File future to the caller. The calling thread can thus continue with its execution, and is subject to a wait-by-necessity on the future to determine if the file transfer has been completed.
3. The file is read in parts by the local FTS, and up to $(o-1)$ simultaneous overlapping parts are sent from the local node to the remote node by invoking $RemoteFTS.savePartAsync(p_i)$ from local FTS [2].
4. Then, a $RemoteFTS.savePartSync(p_{i+o})$ invocation is sent to synchronize the parameter burst, as not to drown the remote node. This will make the sender wait until all the parts $p_i, \ldots, p_i + o$ have been served (ie the *savePartSync* method is executed).

5. The *savePartSync*(...) and *savePartAsync*(...) invocations are served in FIFO order by the remote FTS. These methods will take the part p_i and save it on the disk.
6. When all parts have been sent or a failure is detected, local FTS will update the future created in step 2.

The `pullFile` method is implemented using the *pull algorithm* shown in Figure 7, and is detailled as follows:

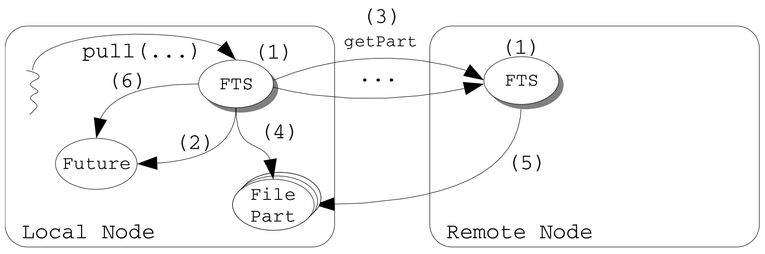


Fig. 7. Pull Algorithm.

1. Two FTS active objects are created (or obtained from a pool): a local FTS, and a remote FTS. The *pull* function is invoked on the local FTS: $LocalFTS.pull(\ldots)$.
2. The local FTS immediately returns a File future, which corresponds to the requested file. The calling thread can thus continue with its execution and is subject to a wait-by-necessity on the future.
3. The $getPart(i)$ method is invoked up to o (internally defined) times, by invoking $RemoteFTS.getPart(i)$ from the local FTS [2].
4. The local FTS will immediately create a future *file part* for every invoked $getPart(i)$.
5. The *getPart*(...) invocations are served in FIFO order by the remote FTS. The function *getPart* consists on reading the file part on the remote node, and as such, automatically updating the local futures created in step 4.
6. When all parts have been transferred, then the local FTS will update the future created in step 2.

4.2 File Transfer after application execution

Collecting the results of a Grid computation distributed in files on different nodes is an indispensable task. Since determining the termination of a distributed application is hard and sometimes impossible, we believe that the best way is to have non-automatic file retrieval, meaning that it is the user's responsability to trigger the file transfer at the end of the application execution (i.e once the application data has been produced).

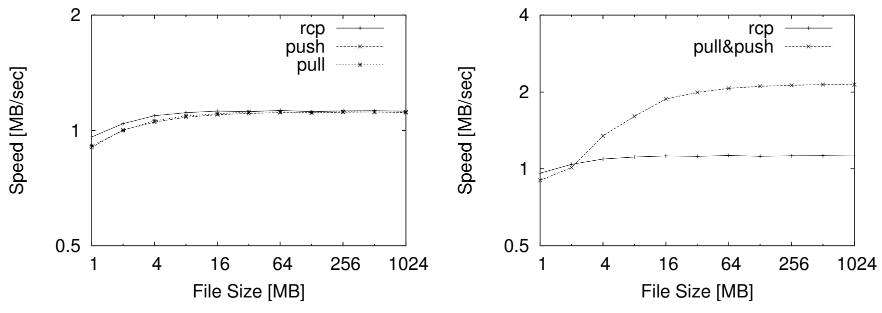
The file transfer retrieval is implemented as part of the API shown in Figure 5. For each node in the `virtualNode`, a `pullFile` is invoked, and an array of futures (`(File[]`) is returned. The retrieved files are the ones specified in the deployment descriptor, as shown in Figure 4.

5 Benchmarks

5.1 File Transfer Push and Pull

Using a 100Mbit LAN network with a $0.25[ms]$ ping, and our laboratory desktop computers: Intel Pentium 4 (3.60GHz) machines, we experimentally determined that overlapping 8 parts of size $256[KB]$ provides a good performance and guarantees that at the most $2[MB]$ will be enqueued in the remote node. The communication protocol between active object was configured to `RMI`.

Since peers usually have independant download and upload channels, the network was configured at $10[\frac{Mbits}{sec}]$ duplex. Figure 8(a) shows the performance results of `pull`, `push`, and *remote copy protocol* (`rcp`) for different file sizes. The performace achieved by `pull` and `push` approaches our ideal reference: `rcp`.



(a) Push, pull, and rcp speed.   (b) Pushing while pulling a file.

Fig. 8. Performance comparisons.

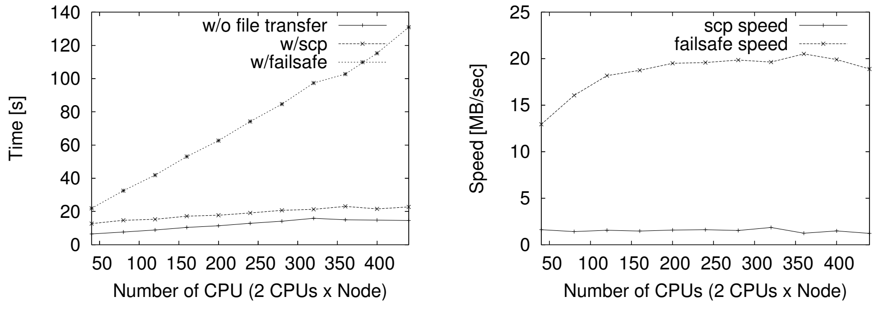
More interestingly, Figure 8(b) shows the performance for getting a file from a remote site, and then sending this file to a new site. This corresponds to a recurrent scenario in data sharing peer to peer networks[11], where a file can be obtained from a peer instead of the original source.

As we can see in Figure 8(b), `rcp` is outperformed when using `pull` and `push` algorithms. While `rcp` must wait for the complete file to arrive before sending it to a peer, the `pull` algorithm can pass the future file parts (Figure 7) to the `pull` algorithm even before the actual data is received. When the future of the file parts are updated, automatic continuation [5, 6] will take care of updating the parts to the concerned peers. The user can achieve this with the API shown in Figure 5, by passing the result of an invocation as parameter to another.

5.2 Deployment with File Transfer on a Grid

Our deployment experiments took place on the large scale national french wide infrastructure for Grid research: *Grid5000* [8], gathering 9 sites geographically distributed over France.

Figure 9(a) shows the time for three different deployment configurations combined with a transfer of a 10[MB] file: regular deployment without involving file transfer, deployment combined with (`scp`), and deployment combined with the failsafe file transfer protocol (which uses the `push` algorithm). The figure shows



(a) Time (b) Speed

Fig. 9. Deployment with 10[MB] File Transfer on Grid5000.

that combining deployment with `scp` adds a constant overhead, while failsafe adds a linear overhead. This happens, because the nodes in Grid5000 are divided into sites, and each site is configured to use *network file sharing*. If the deployment descriptor is configured with `scp`, the file transfer only has to be performed a time proportional to the number of sites used (2 for the experiment). On the other hand, since the failsafe mechanism transfers files from node to node using the file transfer API (of section 4.1), then the overhead is proportional to the number of acquired nodes.

It is important to note, that when using failsafe, the files are deployed in parallel to the nodes. This happens because several invocations of push, on a set of nodes, are eventually served in parallel by those nodes. On the other hand, `scp` transfers the files sequentially to each site in turn. The result is that failsafe reaches a better speed than `scp`, as shown in Figure 9(b), where `scp` averages $1.5[\frac{MB}{sec}]$ while failsafe averages $18[\frac{MB}{sec}]$.

6 Related Work

The importance of file transfer and resource acquisition has been studied, among others, by Giersch et al.[7], and Ranganathan et al.[12], who showed that data transfer can affect application scheduling performance. Solutions for integrating resource acquisition and file transfer have been developed by several Grid middlewares like Unicore[17], and Nordugrid[10]. Our approach differs mainly because it allows on-the-fly deployment while combining heterogeneous resource acquisition and file transfer protocols.

The proposed deployment approach can be seen as a wrapper for third party file transfer tools. Other approaches for using third party tools exist. The main goal behind them is to provide a uniform API. This has been done in Java CoG[18], and GAT[13]. Nevertheless, our motivations differ since we seek on-the-fly deployment, rather than file transfer during application execution.

For transferring files between Grid nodes, GridFTP[1] is a popular tool, which extends the traditional FTP[9]. The approach we propose at the programming level mainly varies from GridFTP because we do not require an underlying file transfer protocol to perform file transfer. On the contrary, we only rely on portable always executable asynchronism with future remote method calls. Therefore, we can benefit from automatic continuation to improve peer to peer file transfer performance, as shown in Figure 8(b).

Concerning the retrieval of files, Unicore[17] and NorduGrid[10] have addressed this issue. Once the job has finished, files generated during the computation can be downloaded from the job workspace using the respective middleware client. Our approach differs because we provide a user triggered API file retrieval mechanism, which allows the user further flexibility. The API can be used by the application at any point during execution once output results are relevant to be transferred, and not only at the very end of the run.

7 Conclusions and Future Work

We have addressed file transfer for the Grid by focusing on three different stages of Grid usage: deployment, execution and retrieval. Our experiments show that it is possible to integrate heterogeneous file transfer with resource acquisition protocols to allow on-the-fly deployment, which can deploy the Grid application and install the Grid middleware at the same time. Experimentally, we have benchmarked the proposed solution, and shown that it is scalable.

For the application execution, we proposed an asynchronous overlapping file transfer mechanism using `push` and `pull` algorithms, built on top of an active object communication model with futures and wait-by-necessity. Experimentally we showed that both can achieve a performance similar to `rcp`. Additionally, we showed how automatic continuation can be used to transfer files between peers in an efficient way.

Finally, we proposed a user triggered file retrieval mechanism for the Grid. This mechanism uses the algorithms developed here in combination with infrastructure information located inside the deployment descriptors.

In the future we would like to explore distributed file systems, built on top of the proposed file transfer API. We also plan to investigate the interaction of file transfer with structured distributed programming models known as skeletons.

References

1. B. Allcock, J. Bester, J. Bresnahan, A. L. Chervenak, I. Foster, C. Kesselman, S. Meder, V. Nefedova, D. Quesnel, and S. Tuecke. Data management and transfer in high performance computational grid environments. *Parallel Computing*, 28(5):749–771, 2002.
2. F. Baude, D. Caromel, N. Furmento, and D. Sagnol. Overlapping communication with computation in distributed object systems. In *HPCN Europe '99: Proceedings of the 7th International Conference on High-Performance Computing and Networking*, pages 744–754, Amsterdam, The Netherlands, 1999. Springer-Verlag.
3. F. Baude, D. Caromel, L. Mestre, F. Huet, and J. Vayssière. Interactive and descriptor-based deployment of object-oriented grid applications. In *Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing*, pages 93–102, Edinburgh, Scotland, July 2002. IEEE Computer Society.
4. D. Caromel. Toward a method of object-oriented concurrent programming. *Communications of the ACM*, 36(9):90–102, 1993.
5. D. Caromel and L. Henrio. *A Theory of Distributed Object*. Springer-Verlag, 2005.
6. S. Ehmety, I. Attali, and D. Caromel. About the automatic continuations in the eiffel model. In *International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA'98*, Las Vegas, USA., 1998. CSREA.
7. A. Giersch, Y. Robert, and F. Vivien. Scheduling tasks sharing files on heterogeneous master-slave platforms. In *12th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2004)*, pages 364–371, A Coruña, Spain, February 2004. IEEE Computer Society Press.
8. Grid5000. http://www.grid5000.fr.
9. J. Reinolds J. Postel. Rfc959 file transfer protocol.
10. NorduGrid. http://www.nordugrid.org.
11. A. Oram. *Peer-to-Peer: Harnessing the Power of Disruptive Technologies*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2001.
12. K. Ranganathan and I. Foster. Decoupling computation and data scheduling in distributed data-intensive applications. In *HPDC '02: Proceedings of the 11 th IEEE International Symposium on High Performance Distributed Computing HPDC-11 20002 (HPDC'02)*, page 352. IEEE Computer Society, 2002.
13. E. Seidel, G. Allen, A. Merzky, and J. Nabrzyski. Gridlab: A grid application toolkit and testbed. *Future Generation Computer Systems*, 18:1143–1153, 2002.
14. INRIA OASIS Team and ETSI. 2nd grid plugtests report. http://www-sop.inria.fr/oasis/plugtest2005/2ndGridPlugtestsReport.pdf.
15. INRIA OASIS Team and ETSI. Second grid plugtests demo interoperability. *Grid Today*, 2005. http://www.gridtoday.com/grid/520958.html.
16. ProActive INRIA Sophia Antipolis OASIS Team. http://proactive.objectweb.org.
17. Unicore. http://www.unicore.org.
18. G. von Laszewski, B. Alunkal, J. Gawor, R. Madhuri, P. Plaszczak, and X. Sun. A File Transfer Component for Grids. In H.R. Arabnia and Youngson Mun, editors, *Proceedings of the International Conferenece on Parallel and Distributed Processing Techniques and Applications*, volume 1, pages 24–30. CSREA Press, 2003.