

File transfer in Grid applications at Deployment, Execution and Retrieval

Françoise Baude Denis Caromel Mario Leyton

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

Abstract

In this article a three staged file transfer approach for the Grid is proposed. File transfer in the Grid can take place at three stages: deployment, user application execution, and retrieval (post-execution). Each stage has it's own environmental requirements, and therefore different techniques must be applied.

The contributions presented in this article can be grouped in two. First, the integration of heterogeneous resource acquisition and file transfer protocols. This allows on-the-fly deployment and retrieval of files. Secondly, an asynchronous file transfer mechanism based on active objects, wait-by-necessity, and automatic continuation.

The proposed file transfer model has been implemented using the ProActive middleware. ProActive provides, among others, a Grid infrastructure abstraction using deployment descriptors, and a programming model based on active objects with transparent futures. Finally, the proposed file transfer model is benchmarked and validated with a real case study: BLAST.

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 retrieval (post-execution). At deployment time, by focusing on integrating heterogeneous file transfer and resource acquisition protocols to allow *on-the-fly* deployment. During the application run time, by offering a parallel and asynchronous file transfer mechanism based on active objects, wait-by-necessity, and automatic continuation. Once the execution of the user application has finished, by offering a file retrieval mechanism.

Document Layout

This document is organized as follows. Background on the Grid programming middleware ProActive is provided in section 2. The background is divided into two parts, the first one explains the active object programming model, and the second one the deployment framework.

The file transfer proposal for the Grid, and how it is implemented in the context of ProActive, is shown in sections 3 and 4. Section 3 describes 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, the notation is introduced and some general concepts concerning resource acquisition and file transfer are reviewed. Section 4 details the file transfer approach used during execution and retrieval, which are built on top of the active object programming model.

Benchmark of the model's implementation are provided in section 5. A

case study application is presented in section 6, where the proposed file transfer model is used to distribute an application to perform sequence alignment in bioinformatics: BLAST [5].

Related work is reviewed in section 7, and finally the concluding remarks and future work are presented in section 8.

2 Background on ProActive

2.1 ProActive Active Objects and Futures

Figure 1, shows the *active object* (AO) programming model used in ProActive [20]. 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* [6].

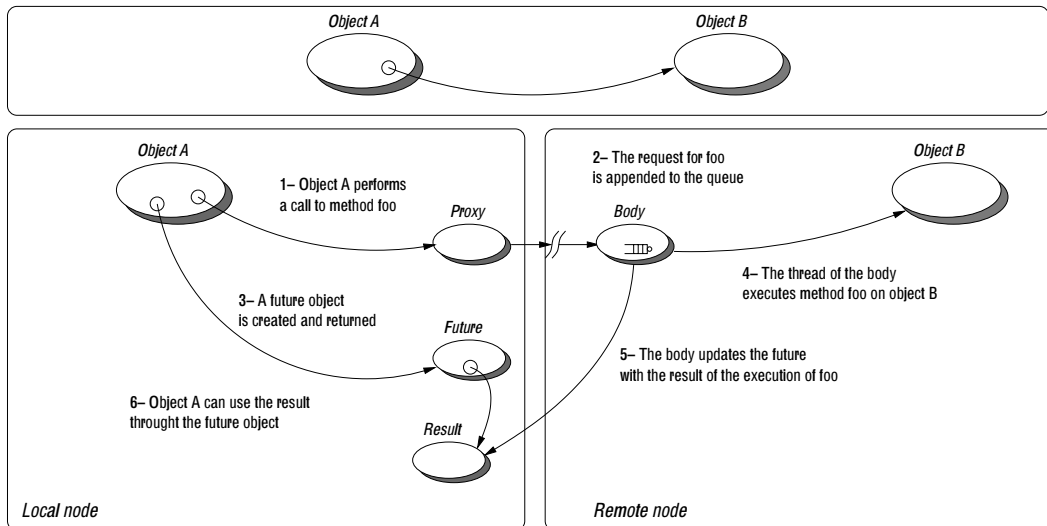


Figure 1: Execution of a remote method call.

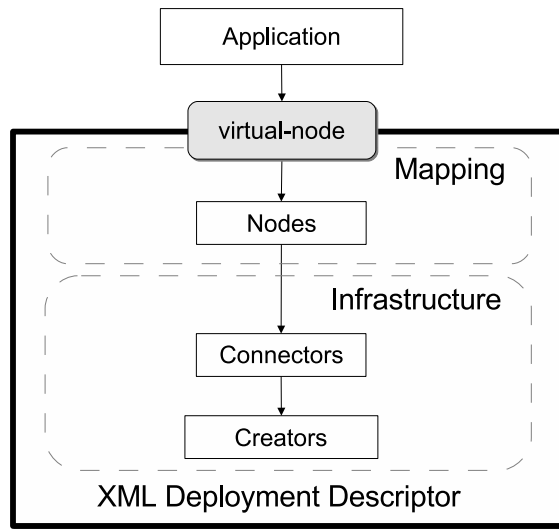


Figure 2: Deployment Descriptor.

2.2 ProActive Deployment Descriptors

ProActive also provides a *Descriptor Deployment Model* [4], which allows the deployment of applications on heterogeneous sites without changing the application source code. All information related with the deployment of the application is described in the Deployment Descriptor (XML). 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/lookup protocols (rmi, jini, http, etc..).

The Deployment Descriptor 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` abstraction is provided, which corresponds to the actual reference in the application code. `Virtual-nodes` have a unique identifier which is referenced inside the application and the descriptor.

The person deploying the application can change the mapping of the Application \rightarrow `virtual-node` to deploy on a different Grid, without modifying a single line of code in the application.

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. 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.

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. Then, a node n_k is acquirable from n_0 iff $\exists\{r_0(p_0), \dots, r_{k-1}(p_{k-1})\}$ and $\exists\{n_0, \dots, n_{k-1}\}$ 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, and will be referred to as a **virtual-node**, using the ProActive terminology.

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

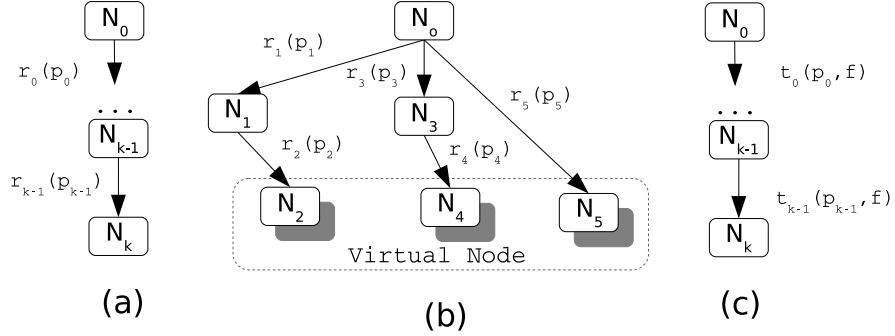


Figure 3: Resource acquisition and file transfer.

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 \vec{t}_k that can or cannot be successful at transferring f from n_k to n_{k+1} . Then, if there $\exists t_k^i \in \vec{t}_k$ which corresponds to the lower indexed transfer protocol capable of transferring f , the sequencing of file transfer and resource acquisition protocols is proposed in the following way:

1. If t_k^i is external, then

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

will be executed. 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

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

will be executed. 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

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

will be executed. In the failsafe approach, the actual file transfer is performed after the resource acquisition.

There are two main reasons 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 consequently must be transferred in advance.

3.4 File Transfer in ProActive Deployment Descriptors

Figure 4 shows how the approach is integrated into ProActive Deployment Descriptors, by taking advantage of the descriptor's structure to apply separation of concerns. The 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 copy the files (`copyProtocol`), hostnames, usernames, etc. Further details of the *failsafe* protocol shown in the example are described in section 4.1. 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).

```

<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>

```

Figure 4: Example of file transfer in Deployment Descriptors.

4 File Transfer during execution and retrieval

Applications can generate data, and transferring this data during the application execution is usually achieved with a specific communication protocol. Nevertheless, Grid resources are characterized by distributed ownership and therefore diverse management policies, as our own experiments [18, 19] confirm it. As a result, setting up the Grid to allow message passing is a painful task. Additionally configuring and maintaining a specific file transfer protocol between any pair of nodes seems to us as an undesirable burden¹.

Therefore, file transfer protocol should be built on top of other protocols,

¹Deployment file transfer does not impose this burden, because the file transfer does not take place between each possible pair of nodes.

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 a message passing model based on active object can be used as the ground for a: portable, efficient, and 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 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

File transfer between nodes has been implemented as service methods available in the ProActive library, as shown in Figure 5. Given a ProActive **Node** *node*, a **File(s)** called *source*, and a **File(s)** called *destination*, the *source* can be pushed (*sent*) or pulled (*get*) from a *node* using the API. The figure also shows a `retrieveFiles` method, which is discussed in section 4.2.

```
//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();
```

Figure 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 detailed as follows:

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(...)*.

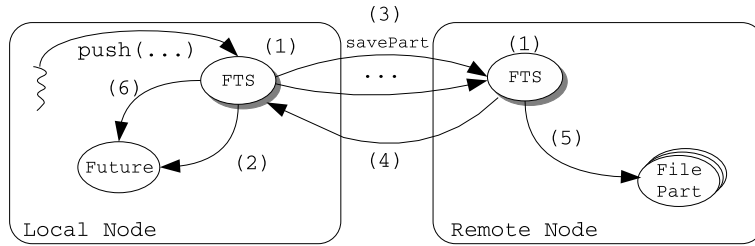


Figure 6: Push Algorithm.

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 [3].
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 $p_i, \dots, p_i + o$ 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 detailed as follows:

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*(...).

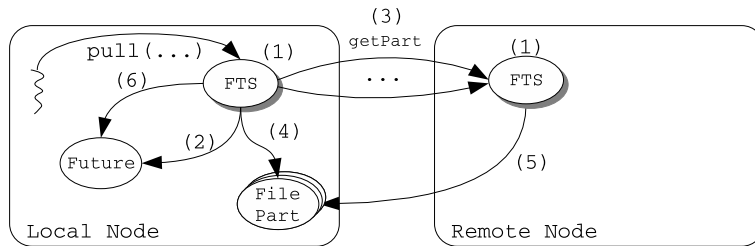


Figure 7: Pull Algorithm.

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 [3].
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 responsibility to trigger the file transfer at the end of the application execution (i.e once the application data has been produced).

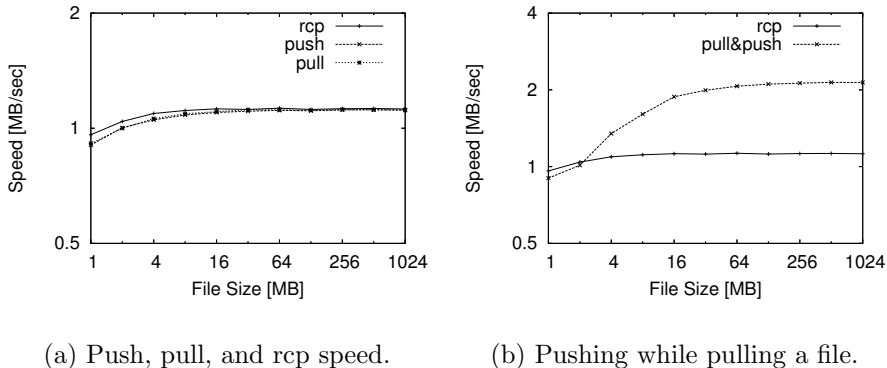


Figure 8: Performance comparisons.

The file transfer retrieval is implemented as part of the API shown in Figure 5. For each node in the `virtual-node`, 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. Because the interest of the experiment was to evaluate the proposed file transfer approach, and not the lower level communication protocols between active objects, the default protocol was used: `rmi`.

Since peers usually have independent 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 performance achieved by `pull` and `push` approaches our ideal reference: `rcp`.

More interestingly, Figure 8(b) shows the performance of 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 [14], where a file can be obtained from a peer instead of the original source.

As it can be seen 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 available, automatic continuation [7, 8] 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

The deployment experiments took place on the large scale national french wide infrastructure for Grid research: *Grid5000* [10], 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 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

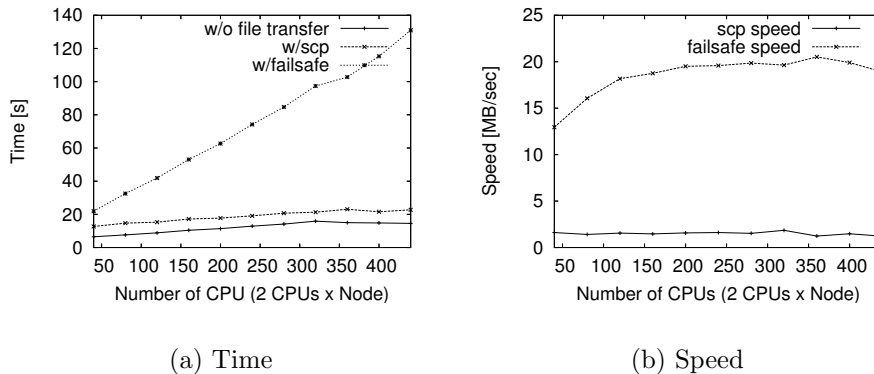


Figure 9: Deployment with 10[MB] File Transfer on Grid5000.

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 Case Study: Distributed BLAST

BLAST [5] corresponds to Basic Local Alignment Search Tool. It is a popular tool used in bioinformatics to perform sequence alignment of DNA and proteins. BLAST is a good case study because it performs intensive file access and computation. In short, BLAST reads a query file and performs an alignment of this query against a database file. The results of the alignment are then stored in an output file.

The approach used for parallelizing BLAST corresponded to splitting the database and distributing it among the computation nodes. Each node is then given the query file to perform a BLAST alignment with its corresponding part of the database. Once the computations are finished, the output file of each node is retrieved and consolidated into a single result.

This experiment took place in Grid5000. The machines involved corresponded to dual core AMD Opteron at 2.2GHz, with 2[GB] RAM. The network bandwidth was configured to $1[\frac{Gb}{sec}]$. The BLAST tool version 2.2.15

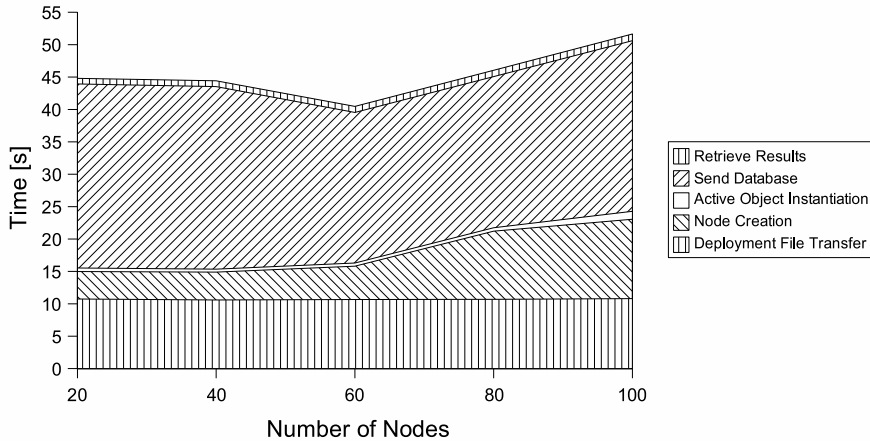


Figure 10: Distributed BLAST overheads.

provided by NCBI [12] was used. As a query, the *ecoli* nucleotide file was selected (5.8[MB]), and as a database the patented nucleotide file (2.4[GB]) was used. Both are publicly available from the NCBI website.

During the deployment phase, the Deployment Descriptor was configured to perform on-the-fly deployment using `scp`. In other words, the ProActive middleware was installed at the same time as the nodes were acquired. Additionally, the query file was also transferred to the computing nodes during the deployment.

Once the resource acquisition (creation of nodes) was completed, a slice of the divided database was transferred to each node using the `push` algorithm. Then, the BLAST tool was executed in parallel on each node using the active object programming model. Once the computation was concluded, the retrieval of the result files was performed using the `pull` algorithm.

6.1 Discussion

Figure 10 shows the overheads introduced when performing a distributed BLAST. The figure shows that the time required to perform the deployment file transfer is independent of the number of nodes, because it is performed a

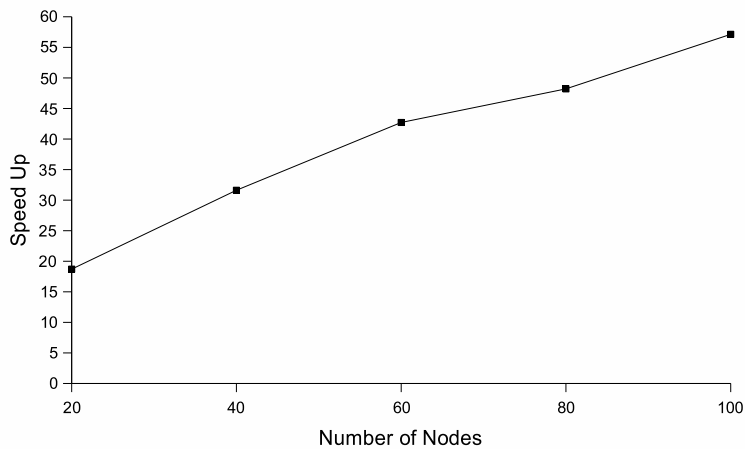


Figure 11: Speedup of BLAST on Grid5000

number of times proportional to the number of sites (one in this experiment). On the other hand, the figure shows that the node creation time is proportional to the number of nodes. Compared with the node creation time, the figure shows that the remote objects instantiation overhead is marginal. The figure also shows that the database `push` experiences performance degradation when transferring to more than 60 nodes at the same time, because of the network saturation, but is still capable of transferring at approximately $100[\frac{MB}{sec}]$.

The speedup of the application is shown in Figure 6.1. The figure shows that, as the granularity increases when using more nodes, the efficiency decreases.

During the BLAST case study, the concepts introduced in this article were used to successfully distribute a BLAST application. On-the-fly deployment was performed to install ProActive and transfer the blast query file at deployment time. The `push` algorithm was used to transfer the database slices to the computation nodes. The retrieval mechanism, in combination with the `pull` algorithm was finally used to retrieve the results from the remote nodes.

7 Related Work

The importance of file transfer and resource acquisition has been studied, among others, by Giersch et al. [9], and Ranganathan et al. [16], 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 [21], and Nordugrid [13]. The proposed approach differs mainly because it allows on-the-fly deployment while combining heterogeneous resource acquisition and file transfer protocols.

The proposed 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 Globus XIO [2], Java CoG [22], and GAT [17]. Nevertheless, the motivations of this work differ since on-the-fly deployment is sought, rather than file transfer during application execution.

For transferring files during application execution GridFTP [1] is a popular tool, which extends the traditional FTP [11]. The approach proposed at the programming level mainly varies from GridFTP because it does not require an underlying file transfer protocol to perform the file transfer. On the contrary, it only relies on portable always executable asynchronism with future remote method calls. Therefore, it can benefit from automatic continuation to improve peer to peer file transfer performance, as shown in Figure 8(b).

LegionFS [23], and MAPFS-Grid [15] provide I/O interfaces for accessing remote data. The proposed file transfer approach does not ambition to be a distributed file system, but instead provides the tools on which one could be built.

Concerning the retrieval of files, Unicore [21] and NorduGrid [13] 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. The proposed approach differs because it provides 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.

8 Conclusions and Future Work

This article has addressed file transfer for the Grid by focusing on three different stages of Grid usage: deployment, execution and retrieval. The 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, the proposed solution has been benchmarked, and shown that it is scalable.

For the application execution, the proposed file transfer approach is based on 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 it has been shown that both can achieve a performance similar to `rnp`. Additionally, it has been shown how automatic continuation can be used to increase the performance when transferring files between peers.

After the application execution, a user triggered file retrieval mechanism for the Grid was proposed. This mechanism uses the algorithms developed in this article, in combination with infrastructure information located inside Deployment Descriptors.

As a case study, it has been shown how the proposed file transfer model can be applied to distribute BLAST, which requires intensive file access and computation.

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.

Acknowledgments

This work was partially funded by CONICYT Chile and the CoreGrid EU Project (FP6-004265).

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] W. Allcock, J. Bresnahan, R. Kettimuthu, and J. Link. The globus extensible input/output system (xio): A protocol independent io system for the grid. In *IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 4*, page 179.1, Washington, DC, USA, 2005. IEEE Computer Society.
- [3] 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.
- [4] 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.
- [5] BLAST. Basic local alignment search tool. <http://www.ncbi.nlm.nih.gov/blast/>.
- [6] D. Caromel. Toward a method of object-oriented concurrent programming. *Communications of the ACM*, 36(9):90–102, 1993.

- [7] D. Caromel and L. Henrio. *A Theory of Distributed Object*. Springer-Verlag, 2005.
- [8] 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.
- [9] 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.
- [10] Grid5000. <http://www.grid5000.fr>.
- [11] J. Reinolds J. Postel. Rfc959 file transfer protocol.
- [12] NCBI. National center for biotechnology information. <http://www.ncbi.nlm.nih.gov>.
- [13] NorduGrid. <http://www.nordugrid.org>.
- [14] A. Oram. *Peer-to-Peer: Harnessing the Power of Disruptive Technologies*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2001.
- [15] María S. Pérez, Jesús Carretero, Félix García, José M. Peña Sánchez, and Víctor Robles. Mapfs-grid: A flexible architecture for data-intensive grid applications. In F. Fernández Rivera, Marian Bubak, A. Gómez Tato, and Ramon Doallo, editors, *European Across Grids Conference*, volume 2970 of *Lecture Notes in Computer Science*, pages 111–118. Springer, 2003.
- [16] 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.

- [17] E. Seidel, G. Allen, A. Merzky, and J. Nabrzyski. Gridlab: A grid application toolkit and testbed. *Future Generation Computer Systems*, 18:1143–1153, 2002.
- [18] INRIA OASIS Team and ETSI. 2nd grid plugtests report. <http://www-sop.inria.fr/oasis/plugtest2005/2ndGridPlugtestsReport.pdf>.
- [19] INRIA OASIS Team and ETSI. Second grid plugtests demo interoperability. *Grid Today*, 2005. <http://www.gridtoday.com/grid/520958.html>.
- [20] ProActive INRIA Sophia Antipolis OASIS Team. <http://proactive.objectweb.org>.
- [21] Unicore. <http://www.unicore.org>.
- [22] 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 Conference on Parallel and Distributed Processing Techniques and Applications*, volume 1, pages 24–30. CSREA Press, 2003.
- [23] Brian S. White, Michael Walker, Marty Humphrey, and Andrew S. Grimshaw. Legionfs: a secure and scalable file system supporting cross-domain high-performance applications. In *Supercomputing '01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, pages 59–59, New York, NY, USA, 2001. ACM Press.

Authors

Françoise Baude is an associated Professor at the University of Nice-Sophia Antipolis. Her main research interests are on distributed object and component oriented parallel and distributed programming. She is currently involved in several EU funded and French funded projects dedicated to Grid computing research.

Denis Caromel is a full professor at University of Nice-Sophia Antipolis and CNRS-INRIA. He is also member of the Institut Universitaire de France (IUF), a multi-disciplinary nation academia that selects a few professors based on the excellence of their research records. His research interests include parallel, concurrent, and distributed object-oriented programming. He was an invited visiting scientist at various universities and research institutes (including Digital System Research Center in Palo Alto, NASA Langley Research Center in Hampton, Virginia, and IBM Tom Watson). He has published more than 70 scientific papers in referred international journals and conferences, and edited 5 volumes of Lecture Notes. In 2005 Springer-Verlag published his monograph called "A Theory of Distributed Objects".

Mario Leyton received his Computer Science Engineer degree, with maximum distinction, from the University of Chile in 2004. Currently he is a PhD student at INRIA Sophia-Antipolis, University of Nice-Sophia Antipolis, and CNRS/I3S. His main research interests are on parallel, distributed and grid computing. In particular, the field of structured parallel programming models.

List of Figures

1	Execution of a remote method call.	3
2	Deployment Descriptor.	4
3	Resource acquisition and file transfer.	6
4	Example of file transfer in Deployment Descriptors.	8
5	File Transfer API.	9
6	Push Algorithm.	10
7	Pull Algorithm.	11
8	Performance comparisons.	12
9	Deployment with 10[MB] File Transfer on Grid5000.	14
10	Distributed BLAST overheads.	15
11	Speedup of BLAST on Grid5000	16