What Is the Price of Simplicity?
A Cross-platform Evaluation of the SAGA API

Mathijs den Burger[1], Ceriel Jacobs[1], Thilo Kielmann[1],
Andre Merzky[2], Ole Weidner[2], and Hartmut Kaiser[2]

[1] Vrije Universiteit Amsterdam
{mathijs|ceriel|kielmann}@cs.vu.nl
[2] Louisiana State University
{andre|oweidner|hkaiser}@cct.lsu.edu

Abstract. The abundance of middleware to access grids and clouds and their often complex APIs hinders ease of programming and portability. The Open Grid Forum (OGF) has therefore initiated the development and standardization of SAGA: a Simple API for Grid Applications. SAGA provides a simple yet powerful API with high-level constructs that abstract from the details of the underlying infrastructure. In this paper we investigate the price that possibly comes with such an API. We discuss the effects on expressiveness and ease of programming, and analyze the performance overhead of three different SAGA implementations (written in Java, Python, and C++) on various middleware. We conclude that SAGA is a good pragmatic approach to make grids easily accessible. The API considerably improves usability and uniformity, but offers a compromise between expressiveness and runtime dependencies. The overall performance of the tested implementations is acceptable, but the strict API semantics require various runtime checks that occasionally cause significant overhead, depending on the underlying infrastructure.

1 Introduction

The amount of different middleware to harvest the computational power and storage capacity provided by grids and clouds has exploded into the face of their users. The multiplicity and complexity of all the available APIs hinders ease of programming and portability of applications. After all, average users of these systems are not programming experts with detailed technical knowledge, but scientists that simply want to run their applications faster and/or analyze larger problems. Learning a certain grid middleware API requires a considerable effort for them. Once mastered, the users are still tied to that specific middleware API, which may change completely with the next middleware release, their next cluster or when collaborating with other grids.

The remedy for this situation lies in a simple yet powerful API for grids and clouds, with high-level constructs that shield programmers of scientific applications from the diversity and complexity of these environments. The Open Grid Forum (OGF) has initiated the development and standardization of such an API, called the Simple API for Grid Applications (SAGA).

Version 1.0 of the SAGA specification is based on a set of use cases [12] and a requirements analysis [11], and was released in January 2008 [6]. Since then, various SAGA implementations have been developed in several programming languages on top of various middleware. The experience gained from these implementations is currently used to refine the SAGA API.

The SAGA API is defined in a language-independent manner. Each implementation language requires a *language binding* that defines what SAGA looks like in that language. A SAGA implementation can be built on top of one specific middleware package, or on top of multiple middleware packages simultaneously. The SAGA implementations examined in this paper fall in the latter category, and allow an application to access many different backends using the same API.

Like every high-level API, SAGA abstracts from lower-level complexity. Such an abstraction may come with a certain cost: possible loss of expressiveness, performance overhead, etc. The goal of this paper is to investigate at what price the SAGA API and its implementations offer a simpler programming interface for various middleware.

The remainder of this paper is organized as follows. Section 2 starts with an overview of the SAGA API and then discusses its advantages and disadvantages. In Sect. 3, we describe three SAGA implementations, discuss their design, and evaluate their performance overhead. Finally, we draw our conclusions in Sect. 4.

2 The SAGA API

SAGA provides an extensible, object-oriented API framework. It consists of a look-and-feel part and an extensible set of functional packages. The look-and-feel consists of the following parts:

Base object:	provides all SAGA objects with a unique identifier, and associates session and shallow-copy semantics.
Session object:	isolates independent sets of SAGA objects from each other.
Context object:	contains security information for middleware. A session can contain multiple contexts.
URL object:	provides uniform naming of (possibly remote) jobs, files, services etc.
I/O buffer:	provides unified access to data in memory, either managed by the application or by the SAGA implementation.
Error handling:	uses error codes or exceptions (whatever maps best to the implementation language)
Monitoring:	provides callback functions for events in certain SAGA objects (e.g., job state changes).
Task model:	allows both synchronous and asynchronous execution of methods and object creation.
Permission model:	lets an application allow or deny certain operations on SAGA objects.

Orthogonal to the look-and-feel are the functional packages, providing the actual functionality of the underlying distributed system. Currently, the set of standardized functional packages consists of:

Job:	runs and controls jobs.
Namespace:	manipulates entries in an abstract hierarchical name space.
File:	provides file access.
Replica:	manages replicated files.
Streams:	provides network communication.
RPC:	allows inter-process communication.

In cloud terms, SAGA represents the IaaS model. The job, namespace, and file package support job submission and file access on reserved cloud resources (e.g. via Amazon EC2 or SSH). A package for the selection and reservation of resources is ongoing work. The complete, language-independent specification of the SAGA API can be found in [6].

2.1 Discussion

We will first describe the advantages of the SAGA API, and then discuss its disadvantages. The three main advantages of the SAGA API are:

Simplicity: the SAGA API is much simpler to use than most middleware APIs. It does not require detailed knowledge of the underlying protocols, but offers a clear set of objects and high-level operations. Programming a grid application is therefore much easier and also comprehensible for non-technical users.

Uniformity: each functional package of the SAGA API is the same for all middleware. For example, file access via GridFTP [1] looks exactly the same as file access via SSH. The same holds for job submission, name space manipulations, etc. The learning curve to access new middleware is therefore removed, and switching middleware (e.g., to boost performance) is trivial. SAGA applications are also very easy to port. Moreover, the SAGA API is very similar across programming languages. The specification of the SAGA API is language independent, and each language binding offers a very similar set of concepts, objects, method names, etc. Once having learned how to use SAGA in programming language A, switching to language B is fairly easy.

Expressiveness: a SAGA implementation often adds functionality that is not supported natively by the middleware or not working in practice. For example, SAGA offers callback methods that notify an application about the status of a submitted job. Various middleware does not provide such callbacks, or the mechanism does not work in practice because of firewalls. A SAGA implementation can then fall back to polling, which may be less efficient but at least works. Other examples are the caching of Globus connections to avoid repeated authentication [15], or the use of Condor's glide-in mechanism for job submission [4].

In general, a SAGA implementation can add any workarounds or performance optimizations available for certain middleware.

We demonstrate the simplicity and uniformity of the SAGA API with example Java code to make a remote copy of a single file. Figure 1 shows the 33 lines of code needed to do this with Globus GridFTP. Figure 2 shows the over 31 lines of code that do the same using the SSH Trilead library [16]. Both examples are reasonably complex, and require detailed knowledge of the underlying middleware (e.g., whether to use *active* or *passive* mode in FTP, or that the stdout and stderr of a remotely executed command have to be read away to prevent deadlocks). Even worse, both examples are *completely different*. Using SSH instead of GridFTP to transfer files requires a user to learn a completely new library, while the essential operation remains exactly the same.

```
1  String host = "host.example.com";
2  int port = 2811;  // default GridFTP port
3  String path = "/home/john";
4  String src = "file.dat";
5  String dst = "newfile.dat";
6  GridFTPClient c1, c2;
7
8  try {
9      GSSManager manager = ExtendedGSSManager.getInstance();
10     GSSCredential credential =
11           manager.createCredential(GSSCredential.INITIATE_AND_ACCEPT);
12
13     GridFTPClient c1 = new GridFTPClient(host, port);
14     c1.authenticate(credential);
15     c1.setType(GridFTPSession.TYPE_IMAGE);
16
17     GridFTPClient c2 = createClient(host, port);
18     c2.authenticate(credential);
19     c2.setType(GridFTPSession.TYPE_IMAGE);
20
21     HostPort hp = c2.setPassive();
22     c1.setActive(hp);
23
24     c1.changeDir(path);
25     c2.changeDir(path);
26
27     c1.transfer(src, c2, dst, true, null);
28 } catch (Exception e) {
29     e.printStackTrace();
30 } finally {
31     if (c2 != null) c2.close();
32     if (c1 != null) c1.close();
33 }
```

Fig. 1. File copy example using the Globus GridFTP API

```
String host = "host.example.com";
int port = 22; // default SSH port
String path = "/home/john";
String src = "file.dat";
String dst = "newfile.dat";

try {
    HostKeyVerifier v = new HostKeyVerifier(false, true, true);
    Connection c = Connecter.getConnection(host, port, v, true);
    Session s = connection.openSession();

    s.execCommand("cp " + path + "/" + src + " " + path + "/" + dst);

    InputStream stdout = new StreamGobbler(s.getStdout());
    InputStream stderr = new StreamGobbler(s.getStderr());

    String error = readStreams(stdout, stderr);

    while (s.getExitStatus() == null) {
        Thread.sleep(500);
    }
    int exitValue = s.getExitStatus();
    s.close();

    if (exitValue != 0 ||
        (error.length() != 0 && !error.startsWith("Warning:"))) {
        System.err.println("Copy failed: " + error);
    }
} catch (Exception e) {
    e.printStackTrace();
}
```

Fig. 2. File copy example using the Trilead SSH library. The method *readStreams()* reads away stdout and stderr in two separate threads and returns the stderr output, but its implementation is omitted to make this example fit on a single page.

In contrast, Fig. 3 shows how a remote file can be copied via Globus GridFTP using the SAGA API. This example contains only 11 lines of code that just express the high-level file copy operation. To copy the file via SSH instead of Globus, a user would only have to edit the *dir* variable at line 1 and change the URL scheme from *globus://* to *ssh://*.

However, we also experienced several disadvantages of the SAGA API:

Runtime dependencies: many middleware systems do not offer all functionality exposed by the SAGA API. Sometimes, SAGA can add such missing functionality itself (one of its advantages), but such workarounds are often simply not possible. For example, if the method *job.suspend()* is invoked for a job submis-

```
URL dir = URLFactory.createURL("globus://host.example.com/home/john");
URL src = URLFactory.createURL("file.dat");
URL dst = URLFactory.createURL("newfile.dat");

try {
    NSDirectory d = NSFactory.createNSDirectory(dir);
    d.copy(src, dst);
    d.close();
} catch (SagaException e) {
    e.printStackTrace();
}
```

Fig. 3. File copy example using the SAGA API

sion system that cannot suspend jobs, a SAGA implementation can only throw a *NotImplemented* exception. Such a limitation will only be revealed at runtime.

To avoid runtime dependencies entirely, the functionality of the SAGA API would have to be confined to the smallest common denominator of all middleware functionality. SAGA's expressiveness would then be severely limited. Instead, the API now contains various operations that can only be implemented on a limited number of middleware systems. Examples include *extended I/O* for files (only available in GridFTP), suspension and restarting of jobs (often not possible), and the creation of symbolic links to name space entries (not present in some backends and programming languages).

Finding out which SAGA functionality *does* work on which middleware requires either detailed knowledge of the underlying middleware (something SAGA explicitly tries to avoid), detailed documentation read by users (which is possible but unlikely), or some trial and error and the interpretation of *NotImplemented* exceptions.

Limited expressiveness: some middleware may offer functionality that cannot be expressed in the SAGA API. For example, SAGA currently does not support the reservations of resources to run jobs on (i.e., nodes or VM images). The SAGA community actively develops new functional packages for the most prominent omissions, but will always lag behind the latest greatest features of specific middleware.

Strict semantics can limit performance: the strict semantics of the SAGA API sometimes hinder an efficient implementation. For example, SAGA currently only supports random file I/O, while streaming I/O can be much more efficient in some cases. Another example is the method *file.copy(target, flags)* that copies a file to another location. The SAGA specification [6] prescribes:

- If the target is a directory, the source entry is copied into that directory.
- A *BadParameter* exception is thrown if the source is a directory and the *Recursive* flag is not set, or the source is not a directory and that flag *is* set.

– If the target lies in a non-existing part of the name space, a *DoesNotExist* exception is thrown, unless the *CreateParents* flag is given - then that part of the name space must be created.
– If the target already exists, it will be overwritten if the *Overwrite* flag is set, otherwise an *AlreadyExists* exception is thrown.
– If a directory is to be copied recursively, but the target exists and is neither a directory nor a link to a directory, an *AlreadyExists* exception is thrown even if the *Overwrite* flag is set.

These API semantics require a SAGA implementation to perform various checks for each invocation of the *copy()* method: does the target exist, is it a file or directory, does the parent directory exist? Depending on the middleware, the underlying infrastructure, and the number of method calls, such checks can be fairly expensive. Yet at application level such checks can be unnecessary, e.g., because the target directory was just created and is therefore known to exist and be empty.

3 SAGA Implementations

The SAGA API has been implemented in several programming languages by several independent institutes. An overview of the various implementations can be found at the SAGA home page [14]. In this paper we will examine three SAGA implementations in more detail: the reference implementation in C++ created at Louisiana State University [10] and the Java and Python implementations created at Vrije Universiteit Amsterdam.

Both the C++ and the Java SAGA implementation have a plugin-based architecture that was pioneered in SAGA's predecessor, the JavaGAT [13]. Figure 4 depicts this design. A small dynamic *engine* provides dynamic call switching of SAGA API calls to middleware bindings (*adaptors*) which are dynamically loaded on demand and bound at runtime. Each implementation includes several adaptors for various middleware packages.

We demonstrate the behavior of an adaptor-based SAGA implementation via the creation of a SAGA *file* object. The constructor of a file gets a URL with the location of the file, and (optionally) some flags that indicate whether to create a new file, or to overwrite an existing one, etc. The SAGA engine then tries to load *all* available file adaptors. Many will fail, e.g., because they reject the URL scheme. For example, an SSH adaptor will reject a URL starting with *ftp://*. The engine remembers the adaptors that succeed, and forwards later method invocations to the first successful one. Later errors trigger the engine to try another adaptor. Only when all adaptors fail, an exception is thrown to the user.

Unlike its C++ and Java counterparts, the Python SAGA implementation does not use Python-specific SAGA adaptors to implement the functionality of the various SAGA packages. Instead, it acts as a wrapper on top of a Java SAGA implementation. All SAGA functionality is therefore available via Python-specific constructs. Internally, all Python SAGA objects use the Java SAGA

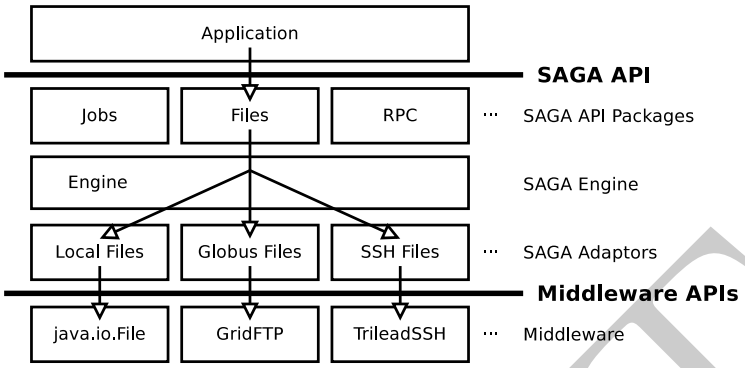


Fig. 4. General architecture of the Java and C++ SAGA implementations: a lightweight engine dispatches SAGA calls to dynamically loaded middleware.

language bindings to implement all functionality. The Python SAGA implementation relies on Jython [9], a Python interpreter written in Java. Jython allows a Python application to use Java objects and methods, which makes it relatively easy to implement Python SAGA on top of Java SAGA.

3.1 Discussion

All three SAGA implementations use or rely on an adaptor-based implementation. The two main advantages of such a design are:

Extensibility: to support new middleware, only new adaptors have to be written. These adaptors can then be easily included in a new SAGA release, or even be added dynamically to an existing SAGA installation. Since SAGA is becoming a standard, one could even envision that middleware developers create and maintain SAGA adaptors themselves.

Flexibility: a single SAGA application can use multiple middleware *simultaneously* with almost no additional effort. To use different middleware, the application only has to provide another URL when creating a SAGA object. Some middleware also requires specific credentials via SAGA context objects. Fortunately, SAGA picks up the default credentials automatically (e.g., SSH keys, Globus certificates, etc.), so specific context objects are usually unnecessary.

However, an adaptor-based SAGA implementation also has some disadvantages:

Error handling is hard: the SAGA engine will try to instantiate *all* available adaptors for a SAGA object. When all adaptors fail, a *compound* exception is thrown that contains a list of all exceptions thrown by the individual adaptors. The compound top-level exception is a copy of the exception in the list that the engine deems the 'most relevant' one. Interpreting such a compound

exception is hard, both in code and for humans. Only examining the top-level exception will ignore the errors of the other adaptors, which may be relevant to understand what is going on. It also requires SAGA to correctly choose the most relevant exception. Examining the whole list instead is more secure but tedious, and requires a user to comprehend why every available adaptor failed. The more available adaptors, the harder is becomes to understand all the errors.

Performance overhead: SAGA adds an additional layer between the application and the middleware, which always introduces a performance penalty. Every additional adaptor increases the initialization time per SAGA object.

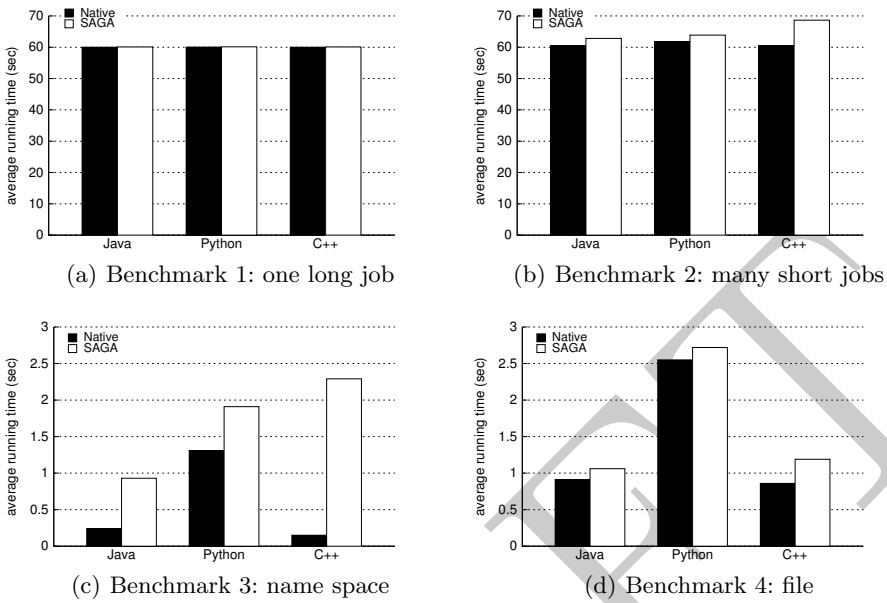
3.2 Performance

To quantify the performance overhead of SAGA, we have created four SAGA applications that benchmark the functional packages *job*, *namespace*, and *file*:

- Benchmark 1 runs one long job (the UNIX command */bin/sleep 60*) and waits until it has finished.
- Benchmark 2 runs a sequence of many short jobs (sixty times */bin/sleep 1*) and waits until they have finished.
- Benchmark 3 executes a series of name space operations:
  1. create 10 directories (*dir000* to *dir009*)
  2. in each directory, create 10 subdirectories (*subdir000* to *subdir009*)
  3. in each sub directory, create 10 empty files (*file000* to *file009*)
  4. recursively print the name, type, and size of all created entries
  5. move all directories *dir** to *d**
  6. copy all files *file** to *f**
  7. recursively remove each directory *d**
- Benchmark 4 performs a sequence of file operations:
  1. create a file *foo* of 100 MB using 3200 write operations of 32 KB
  2. copy file *foo* to file *bar*
  3. read file *bar* in 3200 blocks of 32 KB
  4. remove *foo* and *bar*

We have implemented these four benchmarks on top of the Java, Python, and C++ SAGA implementations and use them in two experiments.

In the first experiment, we compare the *overhead* of each SAGA implementation using only the local machine as the back-end of each benchmark. We compare each benchmark against a native implementation that uses the default language constructs to run local jobs and access local files and directories (i.e., the JDK 1.6, the Python 2.x standard library, and standard C++ together with the Boost Filesystem library. The test machine is located at TU Delft, The Netherlands, and is one of the head nodes of the DAS3 grid system [2]. It features a dual-CPU, dual-core 2.4 GHz AMD Opteron DP 280, 4 GB of main memory and a fast RAID6 storage system. We run each benchmark program 10 times and calculate the average running time. Figure 5 shows the results.



(a) Benchmark 1: one long job

(b) Benchmark 2: many short jobs

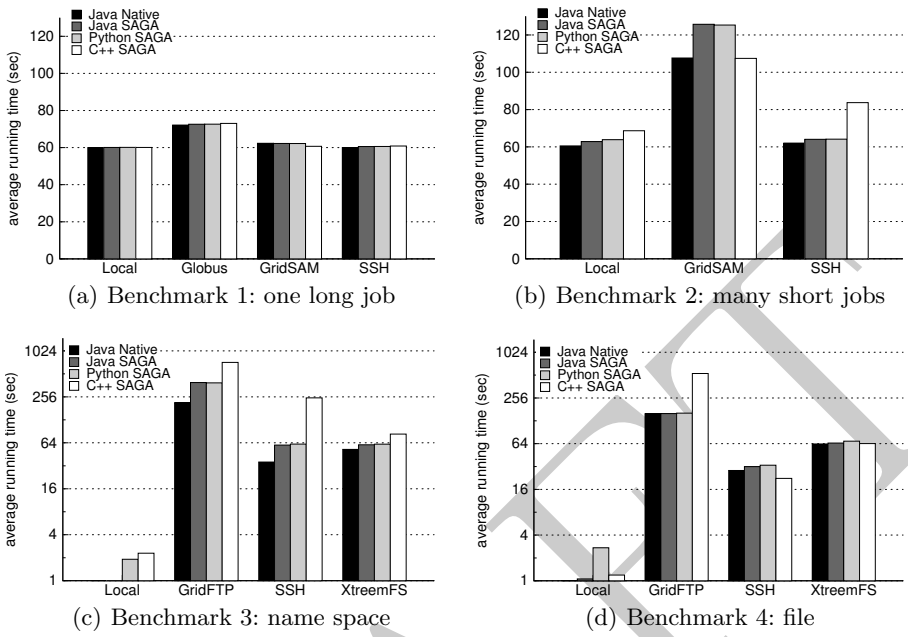
(c) Benchmark 3: name space

(d) Benchmark 4: file

Fig. 5. Average running times of the four benchmarks using SAGA and the native local constructs in Java, Python, and C++.

Running one long job using SAGA takes almost no overhead in all implementations. With many short jobs, the C++ implementation has a slightly larger overhead compared to the Java and Python implementation. The name space benchmark shows a much larger difference between the native and SAGA API, although the overall running time is still quite fast (i.e., less than 3 seconds). The high overhead is caused by all the strict semantics of the SAGA API that requires many additional checks per name space operation. File access speed is comparable in Java and C++, but much slower in Python due to the rather inefficient way Jython handles binary data. Overall, all SAGA implementations show good performance and only moderate overhead. Only with many small operations (e.g., running many jobs or performing many name space manipulations) the overhead starts to show.

In the second experiment, we compare the running time of each SAGA implementation on top of various middleware systems. We also measure the running time of a native Java implementation on top of all middleware to show the overhead of SAGA. We compare local job execution to remote execution using Globus 4.0.3 [3], GridSAM 2.0.1 [7], and SSH. The name space and file benchmarks access the local filesystem and remote volumes via GridFTP [1], SSH and XtreemFS [8]. The local machine in this second experiment is the same as in the first one. All remote machines are located at Vrije Universiteit in Amsterdam, The Netherlands. These two sites are connected by a 1 Gbit link with an RTT of around 3 ms.



(a) Benchmark 1: one long job

(b) Benchmark 2: many short jobs

(c) Benchmark 3: name space

(d) Benchmark 4: file

Fig. 6. Average running times of the four benchmarks using native Java and the three SAGA implementations on top of various middleware.

Figure 6 shows the average running time of 10 repetitions of each benchmark. The overhead of running a single long job is again negligible in all SAGA implementations. Globus needs about 10 seconds to recognize a finished job, which explains its higher baseline in Fig. 6(a). To avoid very long running times, we omitted Globus from the second benchmark with many small jobs. In this case, the C++ SAGA implementation is a bit slower than the Java implementation for local execution and SSH, but faster when using GridSAM. The differences in speed are caused by the different middleware APIs used in the adaptors. For example, the C++ GridSAM adaptor uses a simpler and faster authentication scheme than the Java GridSAM adaptor.

The standard deviation of the running time of most job benchmarks is really small, in the order of 1 second. Only with many small GridSAM jobs the standard deviation gets quite large in Java and Python SAGA. However, this is caused by a memory leak in a security library that is triggered by the (default) authentication mechanism of GridSAM, which significantly slows down the GridSAM server after a while. The simpler authentication mechanism used in C++ SAGA does not trigger this bug, but does require a reconfiguration of the server.

The major speed differences between the SAGA implementations in the name space and file benchmarks are caused by the different implementations of the adaptors. Figure 6(c)s and 6(d) use a logarithmic scale on the Y-axis to accommodate the wide variety in running times. Not surprisingly, all middleware

systems add a considerable amount of overhead compared to the local benchmarks. Especially GridFTP adds a large amount of latency to each operation, which makes the name space benchmark quite slow in all implementations.

In the file benchmark, the performance difference between the C++ implementation and the other ones is caused by the distinct Globus APIs in Java and C++, which come with different performance overheads. The Java GridFTP file adaptor tunes the polling frequency of the GridFTP control channel to minimize the overhead per read and write operation. The C++ GridFTP adaptor relies on callbacks provided by Globus, which slow down each operation.

Besides the random I/O SAGA file API, the Java SAGA language bindings also offer *streaming* I/O since almost all Java I/O is based on streams. We therefore performed the GridFTP file benchmark on top of streaming I/O as well. To our surprise, it then runs an order of magnitude faster than on top of random I/O. We suspect that streaming I/O in GridFTP prevents an explicit acknowledgment of each of the 3200 consecutive read and write operations, which saves an enormous amount of overhead.

The speed differences with the SSH benchmarks are caused by the different middleware APIs used in the adaptors. The SSH adaptor in Java SAGA (and hence also Python SAGA on top of it) uses the Trilead SSH library, while the C++ SSH adaptor uses FUSE [5] to mount a remote filesystem locally and accesses it via the local file adaptor. The former approach is apparently faster for many small name space operations, while the latter is faster for file access. All SAGA implementations use FUSE to access XtreemFS volumes, and achieve comparable speed in both the name space and file benchmarks.

4 Conclusions

The SAGA API offers a simple programming interface for existing grid and cloud middleware. Each functional package of the SAGA API is uniform for all middleware, and very similar across programming languages. SAGA therefore greatly enhances portability and lowers the learning curve for actual users significantly.

On the downside, the SAGA API introduces runtime dependencies in the form of *NotImplemented* exceptions. SAGA may also offer less features than the actual middleware itself, although it can add missing functionality as well (e.g., callbacks of job status via automatic polling). The strict semantics of the SAGA API can sometimes cause significant performance overhead, e.g., in case of many small name space operations or random I/O on top of Globus GridFTP.

For end-user applications, the performance overhead of the tested SAGA implementations is certainly acceptable, and varies with the number of operations, the middleware used, and the underlying infrastructure. We conclude that SAGA's major benefit, its simple and uniform API, largely outweighs the price users have to pay for in terms of runtime dependencies and performance overheads. As such, SAGA has become a viable generic API for grid and cloud environments.

Acknowledgements

Part of this work was supported by the EU IST program as part of the XtreemOS project (contract FP6-033576). Work by Hartmut Kaiser, Andre Merzky and Ole Weidner has been supported by the UK EPSRC grant number GR/D0766171/1 (via OMII-UK) and HPCOPS NSF-OCI 0710874.

References

1. Allcock, W., Bresnahan, J., Kettimuthu, R., Link, M., Dumitrescu, C., Raicu, I., Foster, I.: The Globus Striped GridFTP Framework and Server. In: Proceedings of Supercomputing 2005 (SC05) (November 2005)
2. The Distributed ASCI Supercomputer 3 (2006), http://www.cs.vu.nl/das3/
3. Foster, I.: Globus Toolkit Version 4: Software for Service-Oriented Systems. IFIP Int. Conf. on Network and Parallel Computing pp. 2–13 (2006)
4. Frey, J., Tannenbaum, T., Foster, I., Livny, M., Tuecke, S.: Condor-G: A Computation Management Agent for Multi-Institutional Grids. In: Proceedings of the 10th IEEE Symposium on High Performance Distributed Computing (HPDC10) (August 2001)
5. Filesystem in Userspace (FUSE), http://fuse.sourceforge.net/
6. Goodale, T., Jha, S., Kaiser, H., Kielmann, T., Kleijer, P., Merzky, A., Shalf, J., Smith, C.: A Simple API for Grid Applications (SAGA). Grid Forum Document GFD.90 (January 2008), Open Grid Forum (OGF)
7. GridSAM, http://www.omii.ac.uk/wiki/GridSAM
8. Hupfeld, F., Cortes, T., Kolbeck, B., Focht, E., Hess, M., Malo, J., Marti, J., Stender, J., Cesario, E.: XtreemFS - A Case for Object-based File Systems in Grids. Concurrency and Computation: Practice and Experience 20 (June 2008)
9. The Jython Project, http://www.jython.org
10. Kaiser, H., Merzky, A., Hirmer, S., Allen, G.: The SAGA C++ Reference Implementation. In: 2nd Int. Workshop on Library-Centric Software Design (LCSD'06) (2006)
11. Merzky, A., Jha, S.: A Requirements Analysis for a Simple API for Grid Applications. Grid Forum Document GFD.71 (May 2006), Global Grid Forum (GGF)
12. Merzky, A., Jha, S.: Simple API for Grid Applications - Use Case Document. Grid Forum Document GFD.70 (March 2006), Global Grid Forum (GGF)
13. van Nieuwpoort, R.V., Kielmann, T., Bal, H.E.: User-Friendly and Reliable Grid Computing Based on Imperfect Middleware. In: Proceedings of the ACM/IEEE Conference on Supercomputing (SC'07) (November 2007)
14. SAGA Home Page, http://saga.cct.lsu.edu/
15. Thain, D., Moretti, C.: Efficient Access to Many Small Files in a Filesystem for Grid Computing. In: Proceedings of the 8th IEEE/ACM International Conference on Grid Computing. pp. 243–250 (September 2007)
16. Trillead SSH Library for Java and .NET, http://www.trilead.com/SSH_Library/