Int J Parallel Prog (2009) 37:106–125
DOI 10.1007/s10766-008-0091-4

NetWorkSpace: A Coordination System for High-Productivity Environments

Robert D. Bjornson · Nicholas J. Carriero · Martin H. Schultz · Patrick M. Shields · Stephen B. Weston

Received: 5 March 2008 / Accepted: 7 August 2008 / Published online: 1 January 2009

Abstract As languages and systems for rapid prototyping and application development have grown in popularity and the problems that they are being used to solve have grown in size, so has the need for enhancing them with a coordination facility to support distributed and parallel computations and the creation of application ensembles. We describe NetWorkSpace, a coordination facility based on the fundamental concept of a variable/value binding. NetWorkSpace is implemented as an open source server with clients available for a variety of environments. We present an overview of its design, implementation, performance and usage case studies.

Keywords Coordination · Dynamic programming languages · Distributed computing · NetWorkSpace · Parallel computing · Scripting languages

R. D. Bjornson · N. J. Carriero (✉)
Department of Computer Science, Yale University, P.O. Box 208285, New Haven, CT 06520-8285, USA
e-mail: nicholas.carriero@yale.edu

R. D. Bjornson
e-mail: robert.bjornson@yale.edu

M. H. Schultz · P. M. Shields · S. B. Weston
REvolution Computing, 265 Church St, Suite 1006, New Haven, CT 06510, USA

M. H. Schultz
e-mail: martin@revolution-computing.com

P. M. Shields
e-mail: pat@revolution-computing.com

S. B. Weston
e-mail: steve@revolution-computing.com

1 Conceptual Foundation and Design Goals

Over the last two decades we have worked on coordination languages that augment traditional computational languages (C/C++, Fortran) with facilities to express the basics of coordination: data exchange, synchronization and process creation. As languages and systems for rapid prototyping and application development—sometimes known as dynamic languages and exemplified by MATLAB®, perl, python and R—have grown in popularity and the problems that they are being used to solve have grown in size, so has the need for enabling coordinated computing in these settings as a means to achieve high performance computing (HPC) and to structure computational ensembles. An obvious way to do so is to turn to a coordination system like MPI [1] that is widely used for traditional HPC. One developer of an MPI "add on" for MATLAB estimates that there are at least 20 efforts to effect a marriage of MATLAB and MPI (Kepner J, Private communication). This is indicative both of the growing need for coordination and of the potential difficulties with turning to systems like MPI to meet that need. Users of the systems of interest here have voted with their keyboards for powerful computational abstractions, expressivity and simplicity and are willing to relinquish some performance to achieve them. MPI, in contrast, is designed to enable the realization of very high levels of performance albeit at the cost of potentially complex and intricate coding. MPI's coordination abstraction, such as it is, is too close to the hardware to fit comfortably in environments which try hard to abstract away hardware considerations. Bridging this gap has, evidently, proved to be a daunting challenge.

Our experience with Linda [2,3] argues for a different approach, one based on a simple coordination model exploiting concepts familiar to any programmer and generally applicable to a wide range of problems: name—value bindings. Many programming systems (and all the ones of interest here) have a mechanism for associating (*binding*) a name with a value and for managing sets of such associations. It is so common that most users rarely think about it. A simple illustration:

```
x = 123
y = x*7
print y
```

Here we see bindings of values to the variables 'x' and 'y' with the latter requiring retrieval of the value bound to 'x'. Because manipulating bindings is so common, languages tend to provide syntax to hide the details. The above more explicitly:

```
bind('x', 123)
bind('y', binding('x')*7)
print binding('y')
```

(For brevity we are ignoring how the entities 'bind', 'binding' and 'print' are themselves associated with some interpretation.)

Modern languages have long recognized the need to organize bindings (to simplify naming and avoid unintended references, among other reasons) in what are variously known as scopes, workspaces, environments, etc. Let us assume that we use an OO framework to handle this and that 'ws' is an instance of a class that manages a collection of bindings. The above might then be rewritten:

```
ws.bind('x', 123)
ws.bind('y', ws.binding('x')*7)
print ws.binding('y')
```

(Again, we duck the question of how the binding of 'ws' itself is managed.)

What is the reason for this basic primer? By exposing the mechanics of a foundational concept, we can easily transition to our coordination model by introducing a variant of the binding management class that allows bindings to be shared by any number of processes. The variant encapsulates a client of a shared binding service. Imagine that 'ws' is now an instantiation of this variant. We can change our example to:

```
ws.bind('x', 123)                 # Executed by Process A
ws.bind('y', ws.binding('x')*7) # Executed by Process B
print ws.binding('y')             # Executed by Process C
```

The code is now executed by the cooperative action of three distinct processes. We do not need to change the code per se, just instantiate 'ws' as an object of the shared binding management class and give one line of our simple example fragment to each process for execution.

This illustrates, in a nutshell, the basic concept of a NetWorkSpace. The use of a NetWorkSpace for coordination does induce certain variations in the behavior of bindings, but at its core it is solidly rooted in this foundational concept, and so presents only a modest conceptual step to most users as they transition from sequential to coordinated computing. The essential changes required in the behavior of bindings can be illustrated by further consideration of our example.

First, we have not discussed the constructor for 'ws'. In the simplest case, the null constructor would attach all Processes A, B and C to the same shared binding service under the assumption that all are co-located with the server on the same host. More generally, the constructor would have to include information necessary to identify the service (a host and port).

Second, following the principles of generative communication [2], which lead to simplified coordination via uncoupled and anonymous communication events, referencing a shared binding that has not (yet) been established blocks rather than raises an exception of some sort (or return an uninitialized garbage value) and consecutive bindings don't simply overwrite but instead create a multiset of values associated with the name. Binding a value (using the method `store`) to the variable adds to the multiset (which, by default, is managed as a FIFO queue), while referencing the variable (via the method `fetch`) atomically consumes one value (i.e., one and only process gets a particular value instance). In our example, blocking ensures that we can start Processes A, B and C in any order and still have the code work. Multisets and atomic consumption mean we can easily extend the coordinated code example by wrapping the lines in `while` statements (using a set of different values rather than a single value '123' in Process A) and "the right thing" will happen: we will get a stream of outputs for our stream of inputs.

Finally, even where possible, we do not use a sugared syntax commonly used for normal bindings (illustrated in the first instance of the example). The blocking and

multiset behaviors are distinct from those of a normal binding space. These differences can lead to problems if the shared bindings are used indiscriminately, so it is worth reminding the user of the differences via an explicit syntax. While the default behaviors just described are often a good fit to the coordination needs of an application, they are not always so. Different behaviors are supported: a method that does not consume values (`find`), non-blocking methods (`fetchTry, findTry`), as well as non-FIFO multisets (LIFO, non-deterministic) and single valued modes for variables. Other variants are possible and may be added with time. The choices made thus far result from a pragmatic attempt to balance simplicity with adequate functionality—an attempt informed by early experience with NetWorkSpace and prior experience with Linda-like systems.

As we describe the implementation, we will flesh out this sketch providing details that address a number of pragmatic considerations. We have seen how NetWorkSpace provides two of the basic facilities of coordination: data exchange and synchronization. We conclude this overview of the conceptual foundation with a brief description of the third, process creation. Inspired by Tierney, et al.'s work [4] on *snow* for R, we have added *Sleigh*, a package that offers a parallel "apply" function among other features, to our NetWorkSpace implementations. Imagine that we have a function, `Func`, and a list of argument sets, `argSets`, for `Func`. We want to evaluate `Func` for each of the argument sets. One might use an apply or map operation in a sequential language to do this, e.g.:

```
r = Apply(Func, argSets)
```

The result 'r' is a list whose ith element is `Func(argSets[i])`. If we assume that `Func` is side-effect free, then the computations can in principle be carried out concurrently, e.g.:

```
r = ParApply(Func, argSets)
```

Clearly, for appropriate codes, this is a simple but effective means to substantially improve performance through distributed execution. The Sleigh package provides this capability. Sleigh can also be used to launch components of a more general ensemble, not just evaluation engines for a parallel apply. Because of the close coupling with NetWorkSpace proper, users can coordinate the execution of sleigh processes and exchange data between them to cope with functions that are not side-effect free.

In sum, by providing a variant of the fundamental notion of a binding, NetWorkSpace provides a coordination model that is immediately comprehensible to most users of the systems of interest, and by including a mechanism for orchestrating parallel executions it makes the deployment of distributed applications convenient for these users. It is important to note that this facility complements and extends an existing language/environment. We stop short of suggesting that all variables be transformed into shared bindings. The Symmetric Lisp [5,6] project pioneered these ideas via a principled and thorough design that did encompass all bindings.

Our design goals, however, transcend the development of distributed applications within a particular rapid application development environment. We are interested in creating tools that encourage portability both with respect to coordination idioms and the construction of computational ensembles. Our target users often work with two

```perl
#!/usr/bin/env perl
use strict;

use NetWorkSpace;

my ($Limit, $r, $ws, $x);

$Limit = 7;
$ws = NetWorkSpace->new('camel corral');

for ($x = 1; $x <= $Limit; ++$x) {
    $ws->store('num', $x);
}

for ($x = 1; $x <= $Limit; ++$x) {
    $r = $ws->fetch('result');
    print $$r[0], "\t", $$r[1], "\t", $$r[2], "\n";
}
```

```perl
#!/usr/bin/env perl
use strict;

use NetWorkSpace;

my $ws;
$ws = NetWorkSpace->new('camel corral');

my $x;
while (1) {
    $x = $ws->fetch('num');
    $ws->store('result', [$x, $x**2, $x**3]);
}
```

Fig. 1 Perl NetWorkSpace Example. The code uses two shared variables "num" and "result" to coordinate the execution of the main code (*left*) and a compute server (*right*)

or more environments. We want to create a coordination framework that allows techniques developed in one to be equally applicable to another, and, indeed, to allow codes in multiple environments to coordinate their activities so that the user is free to choose the environment that best handles the various sub-components of a complex problem.

To meet these goals we made a number of design decisions:

(1) As described above, we chose to base our coordination model on a broadly used basic concept.
(2) The underlying protocol needs to be simple and place few demands on the programming environment so that we increase the likelihood that we can develop a client for a new environment.
(3) String data should be handled in such a way that it can be used as a lingua franca for inter-environment communication. In a similar vein, variable and workspace names are simply strings. As such, they do not need to adhere to any one system's naming rules and can be coined dynamically to suit programmatic needs (e.g., a simple form of random access array can be implemented by including the index in the name).
(4) The server providing the shared binding service should be implemented using a widely available tool and be open source.
(5) Process launching should anticipate a number of common scenarios (rsh/ssh, batch queuing systems, as well as a Web-mediated launch mechanism).

We provide details in the implementation discussion that follows, but it is worth noting here that NetWorkSpace client interfaces, at varying levels of fit-and-finish, have been created for Emacs Lisp, Java, MATLAB, perl, php, python, R and ruby. An example of the use of the Perl client appears in Fig. 1. Some clients are open source, others are not. The shared binding service runs on Linux, Mac OS X and Windows. The NetWorkSpace server, python client and R client are available from www.sourceforge.net.

2 Implementation

Our review of the conceptual foundations and design goals for NetWorkSpace implicitly assumed a client-server software architecture for NetWorkSpace. Peer-to-peer

would also be possible—we have built both peer-to-peer and client-server solutions for related coordination systems (Linda and Paradise, respectively), but raises certain discovery and robustness issues that would adversely impact our goal of a simple, easily and nearly universally deployable system. Reflecting on this experience, and experience with python and Twisted [7] acquired in other settings, we made a series of implementation decisions:

(1) Accept the benefits of trading some performance for a substantial reduction in the complexity of the server.
(2) Avoid the complexities of a multi-threaded server.
(3) Stick to a simple "wire" protocol.

In practice: (1) led us to use python and Twisted rather than C to implement the server, (2) to use Twisted's deferred mechanism, and (3) to use a combination of ASCII and "native" serialization for communication between the server and clients.

We now present a high-level overview of the main components of the current NetWorkSpace implementation: the shared binding server, the client (as noted above, available for many environments), a web interface, and sleigh.

2.1 Server

The shared binding server accepts requests from clients to create and manipulate NetWorkSpaces. The core operations are:

- NetWorkSpace (constructor)
  The server instantiates a binding object, associating it with a name provided as an argument to the constructor invocation. The binding object contains a python dictionary that maps from variable names to value sets.
- Store
  Is passed a variable name and a value, which are used to update the binding objects dictionary. The mode of the variable will determine the nature of the update.
- Retrieve
  Is passed a variable name and optional arguments. If the variable name has a value associated with it, the value is returned. Depending on the retrieval variant (`fetch` or `find`) the value may be removed. If there is no associated value, then, again depending on the variant (e.g., `fetch` versus `fetchTry`) the request will be deferred or a null value returned.

The server code implements these operations using instances of four basic classes: *netWorkSpace*, *Value*, *Variable*, and the *Server* class proper.

2.1.1 netWorkSpace

An instance of this class contains:

- A python dictionary that maps from a variable name to a variable object.
- The workspace name.

- Persistence status. (Not discussed here.)
- Ownership status of the workspace.

2.1.2 Value

This class includes:

- A descriptor used, among other things to track whether the value is a string and the environment (MATLAB, perl, python, R,…) that produced it.
- The value proper.
- State of the value.
- Methods to access the value and its state.

Aside from the obvious role of tracking a value, this class encapsulates the notion of a value so that alternate storage mechanisms may be used. The handling of very large values, discussed latter, takes advantage of this encapsulation.

2.1.3 Variable

These objects contain

- The name of the variable.
- The name of the workspace of which it is a member.
- An internal identifier for the variable.
- The "mode" of the variable.
- Lists of clients seeking a value for the variable (one list each for `fetch`ers and `find`ers).
- Values for the variable and an index to track them.
- Methods to `set` and `get` values and to notify all clients waiting for the variable to be bound to a value that they should stop waiting (because, say, the variable is being explicitly deleted and the system is shutting down).

`set` and `get` cooperate to implement the various multiset modes. As mentioned, by default the values bound to a variable are managed as a FIFO queue, but LIFO, non-deterministic, and single valued management schemes are available. The latter more closely follows traditional binding semantics and is very useful for maintaining status information (e.g., a progress counter of some sort). `get` also accepts options to implement `fetch` versus `find` and blocking versus non-blocking (`Try`) behaviors. Since the server is single threaded, a blocking client request must be handled carefully. Twisted provides the concept of a "deferred" operation that is used in this case. The `get` code, noticing that there is no value to satisfy a `fetch` or `find` request, creates a deferred object, registers a callback with it, and places it on a queue associated with the variable. When a value is stored to the variable, the queue is examined and if a deferred is present the callback for it is enabled (*not* invoked). When the current store request processing finishes, Twisted will schedule the callback to run. When the callback runs, the value is sent in satisfaction of the original request.

2.1.4 Server

Contains:

- Status information.
- A dictionary mapping workspace names to netWorkSpace objects.
- Methods including: `referenceSpace`, `setVar` and `getVar`.

`referenceSpace`: implements the semantics of workspace referencing. By default, workspaces are created on the fly as they are mentioned by clients. The first client that mentions a workspace "owns" it in the sense that if the workspace is not persistent, it will be destroyed when the connection closes from the client that created it. The server offers a closely related method, openws, that accepts options to control ownership behavior and the persistence of a workspace.

`setVar` and `getVar`: given a workspace name and a variable name find the corresponding variable object (creating it if necessary) and then invoke that object's `set` or `get` method.

In addition, the server class offers methods to explicitly declare a variable (and in particular, its mode), delete a variable or workspace (often used as a sort of out-of-band communication event that causes a client to raise an exception), list workspaces and variables, create a unique new workspace (important for modularity), as well as a variety of other utility methods.

The core code for the NetWorkSpace is ∼600 lines, and for the protocol 300—remarkably small relative to our implementations of similar systems using C. In our introductory remarks on the implementation we briefly touched on client-server versus peer-to-peer implementation strategies. The Twisted client-server implementation is sufficiently lightweight that it is possible to use it in an ad hoc peer-to-peer manner. One process could create a server on the fly. The contact information for this new server is relayed via a variable held by an existing server to other processes, which then use this information to connect to the new server, creating a "direct" communication channel to the first process. One might to do this to improve performance, increase memory available for value storage, or tighten the relationship between an application and a server.

2.2 Clients

By design, the client code is "lean and mean": It does very little other than relay arguments to the server and accept responses. This exchange is done using a very simple communication protocol:

- Four bytes holding an argument count (in decimal ASCII, so a max of 9999 arguments). Note that this is a count internal to our protocol, it does not impose a user-visible limit.
- For each argument:
  - 20 bytes holding a length (in decimal ASCII—this is sufficient to hold a 64 bit count)
  - The argument proper

ASCII representations of counts are used to ease debugging and to avoid endian issues. One critical component here is the creation of a byte stream representation of the argument in the case where the argument is the value for a store operation. We assume that the environments of interest offer a serialization service capable of representing a data structure as a byte stream that may later be used to reconstitute the data structure in a different process (of the same programming environment). So, for example, when a user of the python client executes:

```
ws.store('x', (1.23, {'cat': 'meow', 'dog': 'woof'},
   range(10)))
```

the client code uses python's pickle module to create a byte stream that represents a three field tuple containing a float, a dictionary and a list of ten integers. The server (even though it is itself a python code) does not attempt to do anything with these bytes other than store them. When this client or another python client later fetches this value for 'x', the client code will be sent this byte stream by the server. It will then use the pickle module to recreate the original three field tuple. Serialization is crucial to working conveniently with complex data types, but we do not insist on it. Some client environments may not offer it, others may have limitations that restrict its use. As a fall back, strings are handled directly with no serialization. This means that such strings can be used when no serialization is available, serialization does not work well, or it is desirable to exchange data between disparate environments. This approach does require the user "to roll your own" code to effectively represent the data as a string, but most of the environments of interest have powerful string manipulation capabilities, so this is generally not too great a burden. It is also the case that in many of the environments a "string" is a fairly flexible concept that is in effect simply a byte buffer of a given length. For these environments, it is possible to exchange binary data as a string value if both endpoints can handle the binary format. We use this, for example, to send images generated by R as binary data (in png format) to a python monitor application. Finally, we note that some values can be extremely large. To cope with such values we make use of the Value encapsulation discussed briefly above to allow the server to store extremely large values in memory mapped files. When the server detects that a value is bigger than a specified threshold, it will stream the value data to a file to avoid consuming too much memory. It will also stream the value straight from the file when sending it to a client. When the value is consumed via a `fetch` operation, the file is deleted.

Client requests generally include a short ASCII operation string as their first argument and then additional arguments as appropriate. For a store, these additional arguments would be the workspace name, the variable name, a value descriptor (a code that indicates the environment of origin, and a flag indicating if the value is a simple string) and the value to be stored.

Replies from the server generally include a status value as their first (and often only) argument. A reply to a fetch request would have a second argument containing a value (a string or serialized data), unless some sort of exception occurred in which case the status would reflect this.

2.3 Web Interface

Twisted is designed to be a generic framework for internet applications. One the most common internet applications is some form of web server, so it is not surprising that Twisted offers excellent support for web services. We made use of this support to create a web interface for the NetWorkSpace server. This interface provides monitoring and management of the workspaces held by a NetWorkSpace server, as well as the variables stored in these workspaces. Figure 2 provides screen shots of this interface.

The implementation proper was quite straightforward. Excluding HTML templates, the web interface is ∼600 lines of code. Integration with the NetWorkSpace server was simply a matter of passing a reference for that server (and so for all of the shared binding state as well) to the constructor of the web interface service, registering both services (on different ports) with Twisted's main execution loop and then running that loop. Twisted's infrastructure handles the multiplexing of traffic to the two ports. Because it is single threaded, hazards in sharing the state between the web interface and the NetWorkSpace server are much reduced.

One interesting feature of the web interface is its ability to display values bound to a variable. Recall from our discussion of the client-server protocol that the NetWorkSpace server generally receives values in serialized form. It does not attempt to decipher this form. So when a user asks via the web browser to see the value(s) bound to a variable, we need somehow to translate the values back into "real" data and then present that in some conventional ASCII representation. The web interface proper could attempt this but it would almost certainly be limited to python clients (deserialization of data from other environments would be quite complex and fragile). We want the web interface to support *all* of the various clients. We accomplished this by creating a simple translation service interface: the web interface, using standard NetWorkSpace operations, binds values to be translated to a variable being monitored by a translation process. The translation process is little more than the equivalent of the following code:

```
while 1: ws.store('translated', disp(ws.fetch
         ('translate'))
```

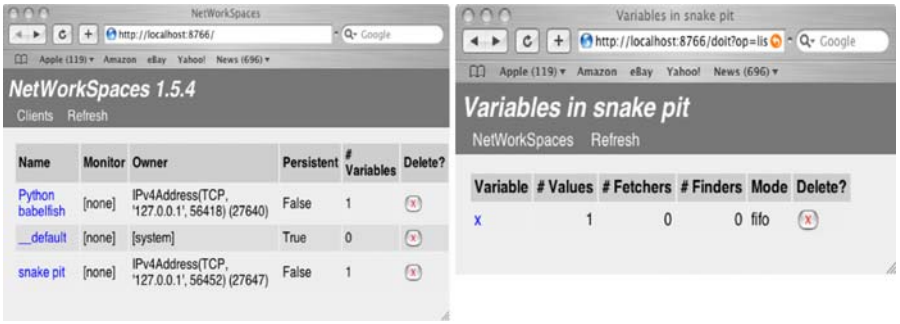

Fig. 2 *Left* list of NetWorkSpaces. *Right* list of variables

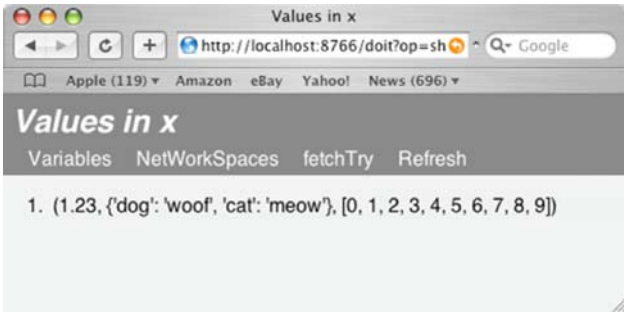

Fig. 3 Web interface display of a value of a NetWorkSpace python variable

running in an appropriate environment (MATLAB, python, R,…) and 'ws' is a reference to a workspace specific to that environment (keeping the translation tasks separate). `disp` is the environment's normal routine for generating a string representation of a value. Because the result bound to the variable `translated` is a pure ASCII string regardless of the environment of origin for the value, it is handled as-is and can be massaged into HTML by the web interface, which `fetches` values for the variable `translated`. These translation processes are called babelfish and we instantiate one per environment of interest. A small table is used to map from the environment of origin for a value (indicated in the value's descriptor) to the babelfish workspace for that environment. If an environment's babelfish is not running, the system notes this and the values from it will not be translated. We note that since the translation process could be time consuming due to the number of values bound to a variable, the complexity of the data, or both, it is done via Twisted's deferred mechanism so that the main server loop will not stall during the translation process. Figure 3 provides an example.

2.4 Sleigh

As one might expect, sleigh makes heavy use of NetWorkSpace. Aside from subprocess (*worker*) start up, sleigh is really just a pre-packaged NetWorkSpace meta-application. Subprocess start up depends on the OS (or mix of OSes—subprocesses can, in principle, run on a variety of platforms) and resource management facilities available. For simplicity, let us assume that we are using a multi-core machine and that all processes will be run locally. By default, instantiating a sleigh object triggers the creation of three worker subprocesses of the same sort as the invoking environment (R creates R subprocesses, python creates python, and so on). These subprocesses in turn each fork into two processes: one monitors the other as well as the overall state of the sleigh computation, while the other carries out sleigh computations proper. The main role of the monitor is to wait for a shutdown indication, which will alert it to kill the computation process. The main role of the computation process is, of course, to carry out sleigh tasks. The various processes here are invoked via shell (or .bat) wrappers, making it easy to customize their behavior and to log their actions (for debugging or "liveness" monitoring purposes). Figure 4 illustrates the general mechanism. The

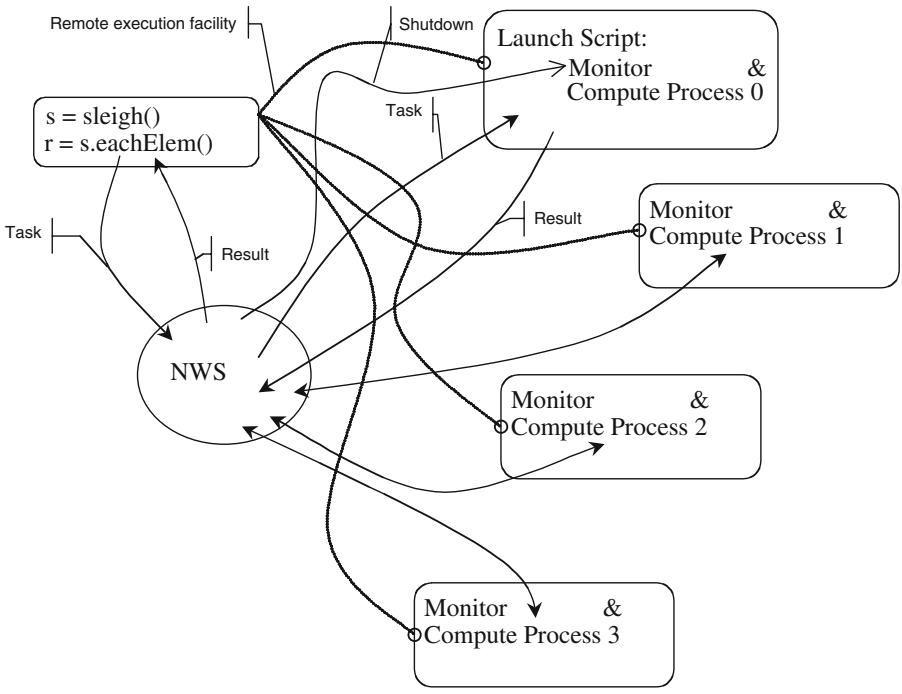

Fig. 4 Sleigh launch and evaluation mechanism

complexity of this figure is indicative of the degree to which the user benefits from sleigh taking care of worker start up and management.

The default configuration of three local workers is useful for development and may be all that it is needed if there is a good match between problem size and local hardware resources, but clearly more flexibility is needed. When a sleigh constructor is invoked, it can be passed a number of options including a list of nodes to use and a mechanism for launching processes on those nodes. Currently supported mechanisms include ssh, rsh, batch queue submission using the Load Sharing Facility system, and a web launch.

The web launch mechanism is a bit unusual. It makes use of the web interface to control launching. The sleigh constructor instantiates a variable specific to the sleigh being created. The variable's value is an ASCII string containing commands for the host environment. To add a worker to the sleigh, the user browses to the NetWorkSpace web interface from the machine on which the worker will run, finds the variable, and copies the command string. The string is then pasted into a manually started instance of the environment being run (MATLAB, python, R,…). The execution of the commands enrolls that instance as a computation worker for the sleigh. When all desired workers have been added in this way, the user deletes another variable via the web interface. This signals to the constructor code that it may complete the instantiation of the sleigh. You would not want to boot a 1000 node cluster this way, but this is a simple and effective solution for "booting" an ad hoc cluster of office machines or student laptops that do not have a remote execution facility enabled.

The sleigh object returned by the constructor provides two methods for execution: `eachWorker` and `eachElem` (analogous to clusterCall and clusterApplyLB in snow). `eachWorker` takes a function argument that is executed once on each of the workers. It returns a list of results when all the function evaluations are complete, one result for each worker in the sleigh. A FIFO-mode "task" variable is assigned a structure representing the function invocation, one structure per worker. The worker processes of the sleigh are simple while loops that `fetch` a value from this variable and execute the encoded function (under a try/except construct or something similar if available). In the case of `eachWorker`, the task includes a flag that instructs it to wait at a barrier after the task is completed until all workers have carried out their function evaluations. This ensures that one and only one function evaluation is done by each worker per `eachWorker` invocation. `eachWorker` is typically used for two purposes: (1) to initialize workers, e.g., import needed libraries or data sets, and (2) to create an ensemble of distinct processes by running a function that selects an ensemble role for each sleigh worker. This function, for example, might make one sleigh worker an input process, another an output process, and the rest compute processes based on rank (available as a global to each sleigh subprocess) or via the manipulation of a NetWorkSpace variable (each sleigh has a "scratch" workspace associated with it, which is made available to each worker via another global variable).

`eachElem` is the sleigh method used to accomplish a ParApply, mentioned earlier. Its implementation is similar to `eachWorker`, but differs in important ways. First, the method takes extra arguments that are used to generate a list of the input sets at which the function will be evaluated. In order to accommodate calling conventions of functions that were written without sleigh in mind, the argument specification is quite flexible. It allows for convenient expression of fixed arguments (values that are the same for all the function invocations) and for the permutation of fixed and varying arguments to match the prototype of the function being invoked. In many cases, this eliminates the need to write a wrapper function to map arguments from an ordering imposed by the parallel apply to the "natural" order of the function (or the related approach of creating code to massage a collection of "natural" inputs into a list of argument sets appropriately formatted for `eachElem`). Here is a python example:

```
>>> from nws import sleigh
>>> s = sleigh.Sleigh()
>>> def testFunc(v0, f0, v1): return (v0, f0, v1)
...
>>> r = s.eachElem(testFunc, [range(0,3), range(3,6)],-1)
>>> r
[(0, 3, -1), (1, 4, -1), (2, 5, -1)]
>>> r = s.eachElem(testFunc, [range(0,3), range(3,6)],-1,
argPermute=0,2,1])
>>> r
[(0, -1, 3), (1, -1, 4), (2, -1, 5)]
```

In the case of an extremely long list of input sets, rather than flood the shared binding server with the full set of tasks, a "flow control" option may be given to issue tasks only as earlier ones are completed.

The second major difference compared to `eachWorker`: the barrier is no longer invoked by each sleigh worker after each task, so each sleigh process can chew through tasks at its own pace. If the ratio of tasks to sleigh workers is reasonably high this tends to lead to fairly good, "automatic" load balancing (although a long last task will still hurt overall efficiency—it is best, if possible, to place input sets requiring more computation early in the list). Given that many of the environments of interest are interactive, an option is also available to make `eachElem` non-blocking. If this option (incompatible with the flow control option) is specified, `eachElem` immediately returns a sleigh pending object. This object may be used to check the status of the computation, to wait for final completion and to retrieve the results.

The python sleigh client also offers the methods imap and starmap (similar to functions of the same name in python's itertools modules). These return a python iterator. As results are returned by the iterator, new tasks will be submitted to the workers for execution. Iterators make it possible to work with extremely long lists of input sets without having to fully instantiate either the input list or the result list. This does come at the cost of potential worker starvation if an iterator is not "next"ed with sufficient frequency.

Finally, the underlying code that implements sleigh leverages the web interface to offer a simple monitoring capability by creating variables whose values track information related to the sleigh's workers. We have also experimented with environment specific monitors. Figure 5 provides an example.

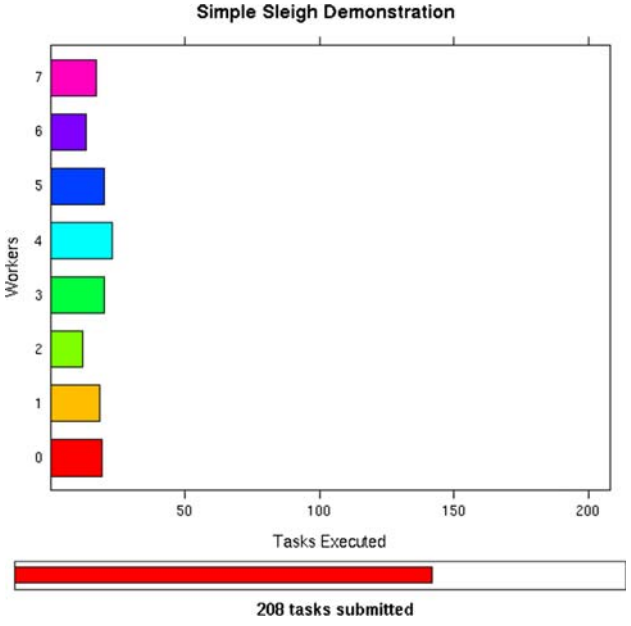

Fig. 5 Sleigh execution monitor: The upper bars indicate the number of tasks done by each worker. The lower progress bar indicates number of tasks done relative to the total number to be done

We will present application-level performance data in the next section. Here we provide some basic data to calibrate expectations. The testing platform consisted of two nodes, allocated for our exclusive use, of a large, heavily used cluster (so some background network traffic is inevitable). Each node had dual 3.0 GHz Intel dual-core Xeon EM64T processors. The nodes were attached to a switched 1 gpbs network. The test consisted of timing the execution of `eachElem` applying the identity function to a list of 100,000 integer inputs. For the first run, the control process, a sleigh worker process and the NetWorkSpace server were co-located on one node (with four cores). Execution time was 92 s. The second run used three workers and ran in 71 s. The third and fourth runs were similar to the first two except that the worker processes were placed on the second node. The execution times were 116 and 81 s, respectively. Given that the evaluation of each task, in effect, executes two fetch/store pairs of modest size values, these times are also indicative of basic communication performance. Note that the times varied considerably from one trial of a run to the next; we report the lowest of three trials. This variance and the timings proper deserve more attention then we give them here, where our intent is simply to provide a sense of coordination costs.

3 Case Studies

We discuss two examples to illustrate the kinds of problems for which NetWorkSpace has been used, as well as to give some idea of representative performance improvements and the general character of the task of developing a parallel application using NetWorkSpace. We will not focus here on the detailed analysis of performance trade-offs and tuning strategies.

3.1 K-means

Clustering (or unsupervised learning) is an important data mining technique. Given a (large) set of unlabelled examples, we wish to automatically determine which examples are alike. K-means [8] is one of oldest and best known algorithms for clustering. There are many popular variants of k-means including: Hartigan-Wong, Lloyd, and MacQueen [9]. In general, initial guesses are made for the centroids of k clusters. The algorithm then loops over the examples, finding the closest centroid for each example (to which the example will be assigned). Once all the examples have been assigned, new centroids are computed for each cluster and the process is repeated until no example changes clusters or numerical convergence is reached for an appropriate scoring function that measures the relative values of intra-cluster and inter-cluster distances for each example.

Unfortunately, it is well known that the k-means algorithm typically converges to local minima of such scoring functions. Typically a global minima is achieved by restarting the k-means algorithm many times with randomly chosen initial guesses for the k centroids and taking the best of the crop. For large datasets, such as those found in document clustering, bioinformatics, marketing, and collaborative filtering applications, this can be very compute intensive.

```
kmeansNWS <- function(x, centers, iter.max = 10, nstart = 400,
       algorithm = c("Hartigan-Wong", "Lloyd", "Forgy", "MacQueen"),
       psleigh)
{
   Sanity check and process the input arguments

   # Store the input matrix to the user workspace where it will
   # be retrieved by each of the workers
   nwsStore(psleigh@userNws, 'x', x)

   # Call each worker to do an appropriate fraction of the total work
   nstart <- ceiling(nstart / workerCount(psleigh))
   r <- eachWorker(psleigh, worker, nstart, centers, iter.max, algorithm)

   Find and return the best result
}

# The worker function gets the input matrix from the nws server
# and calls the kmeans function.
worker <- function(nstart, centers, iter.max, algorithm) {
   kmeans(nwsFind(SleighUserNws, 'x'), centers, iter.max, nstart, algorithm)
}
```

Fig. 6 Parallel k-means code outline

We can think of the sequential k-means algorithm as a triply nested loop. The outer loop ranges over the set of guesses for the initial k centroids. The middle loop iterates over the recomputed centroids. The inner loop ranges over the examples, assigning each to a cluster. From this point of view, coarse grain parallelism can be realized by breaking up the outer loop, while fine grain parallelism can be achieved by splitting the inner loop. (The middle loop is effectively sequential.)

A code outline written in R for the coarse grain approach is given in Fig. 6. For those readers unfamiliar with it, the R programming language [10] offers a cross-platform environment for rapidly implementing mathematical and statistical methods. It is a GNU project that provides a free and open implementation of the S programming language originally developed by AT&T. The code uses R's standard kmeans function, which is a "driver" that can be used to execute a number of k-means variants. The code in Fig. 6 is meant to be simple and readily understandable. It does not address issues like appropriate interactions with the random number generator to ensure reproducibility of sequential results or running exactly nstart evaluations.

To give a sense of the performance gains possible, we ran tests of the Hartigan-Wong variant that used 10 centers and allowed 100 iterations. nstart, the number of random starting points that should be chosen, was set to 400. For the coarse grain parallelization, these starting points are divided among the workers, so that with 8 workers, each worker would evaluate 400/8 sets of starting points. The test data set had 10 columns and 40,000 rows. The tests were run on a dual-processor, quad-core 1 GHz pre-release AMD Barcelona single-board machine. The timing results are presented in Table 1. It should be noted that this and the other codes we discuss here could be run unmodified on other parallel hardware.

We implemented the fine grain approach by splitting the examples among several worker processes. Each worker executes one iteration of a k-means algorithm using R's kmeans function applied to just its examples. Since the fine grain approach

Table 1 K-means parallel performance (coarse)

Workers	Time (s)	Speedup
Sequential	601	
2	312	1.9
4	159	3.8
8	82	7.3

Table 2 K-means parallel performance (fine)

Workers	Time (s)	Speedup
Sequential	274.9	
2	146.1	1.9
4	77.9	3.5
8	37.1	7.4
16	23.8	11.5

involves tinkering with the internals of the k-means algorithm, it is better suited to some variants than others. We used the Lloyd variant for these tests. The workers in this parallel version exchange their partial centriod results, so that each can compute the new centriods. They then call kmeans again using the new centriods for the next iteration, and continue the process until the problem converges.

Typically, one would tend to favor coarse parallelism, but the fine grain approach has an important advantage: it allows each worker to operate on a subset of the data and so reduces the amount of memory required per worker (at the cost, of course, of the need for data communication after each inner-loop iteration). For a large problem, this approach may in fact be the only practical way to run the problem at all.

The tests of this version were run an SGI Altix 450 with 16 1.6 GHz cores. We followed the approach in Dhillon and Modha [11] to generate a large synthetic data set appropriate for this parallelization approach: a matrix with 2^{20} rows and 32 columns. We specified the number of clusters (k) to be 64, one set of starting centroids and the maximum number of iterations to be 10. The algorithm always ran for the full 10 iterations. Timing data is given in Table 2. (The data presented is relative to a sequential execution that executes in "chunks" mimicking the parallel version's data referencing, without this change the speedup would be superlinear.)

3.2 CaretNWS

We now consider a second data mining example: a high performance automated system to classify the toxicity of proposed small molecules for drugs. This example is motivated by the need of the pharmaceutical industry to shorten the time and reduce the cost of developing new drugs. Developers often apply a "supervised learning" data mining approach to the design of such systems. This approach involves "learning" a software classifier (a "machine learning" mathematical/computational model) by training on a database of "gold standard" cases that been previously classified by

domain experts. Once such a classifier has been learned, the model can be used to classify new cases without resorting to experts for manual classification.

R has an extensive system for developing packages that compartmentalize similar types of calculations. *Caret* [12] is one such package. It provides a unified interface to over 25 different types of models from 20 of R's many separate packages for machine learning and predictive modeling.

We will use Caret to evaluate two well-known classification models, support vector machine (SVM) [13] and RandomForests [14]. SVMs attempt to find a prediction rule that best maximizes the distance between two sets of data. This model uses some mathematical techniques to greatly expand the effective dimensions of the predictors in non-linear ways (such as using a radial basis kernel function). RandomForests is a statistical model that builds an ensemble of classification trees. For each node of each tree, the algorithm tries to find a variable to partition the subset of training data at that node into two subsets that are more homogenous. RandomForest models have the feature that, at each split, only a random subset (of a size given by the parameter *mtry*) of variables are considered as possible split variables. The ensemble of trees classifies new cases by "majority vote."

In the process of training these types of models, tuning parameters must be adjusted that cannot be directly estimated from the training set. In caret, the optimal tuning parameters are chosen by trying different candidate values and assessing model performance. One method for assessing performance uses the bootstrap technique: generate many different *resample sets* of the training set (new sets randomly drawn from the training set with replacement), build a model for each resample set, predict the samples that are not in the resample set, then use these predictions to estimate model performance for a particular candidate value of a tuning parameter.

The RandomForest method in caret has a single tuning parameter, mtry (mentioned above). The SVM model has two tuning parameters. The first is a cost parameter that controls the complexity of the prediction rule. As the cost parameter is increased, the SVM model will become increasingly intolerant of misclassified samples and will become more complex. A model that is too simple will not predict the samples well, but a highly complex model may over-fit by adapting to nuances of the training set that do not exist in other samples. The other tuning parameter controls the non-linearity in the radial basis kernel function. It is the exponent's coefficient in the exponential used in the kernel function, and so determines how steep the exponential is. There are no algebraic formulae that can be used to analytically choose these values. The resample approach provides a data-oriented method to select values that avoids under- and over-fitting.

Each calculation for each value of the tuning parameters is independent of the others, suggesting a straightforward parallelization. Table 3 presents performance data for a code that used this approach to parallelization. These tests were run on the AMD Barcelona system. In order to obtain reasonable sequential execution times modifications where made to the base caret package and the code run using numactl to avoid idiosyncrasies of the scheduler (without these changes, the parallel version would have exhibited superlinear speed up).

We have also used NWS to implement a parallelization at a slightly finer level based on the fact that the evaluation of a bootstrap resample set is independent of any other.

Table 3 CaretNWS parallel performance

Algorithm	Workers	Time (s)	Speedup
SVM	Sequential	602.1	
SVM	4	217.2	2.8
SVM	8	118.9	5.1
RF	Sequential	495.9	
RF	4	139.0	3.6
RF	8	73.4	6.8

This version is publicly available as the package caretNWS via CRAN. Speed up for a similar problem using four cores of a 2×4 2.4 GHz Intel based machine was 2.8 for SVM and 3.4 for RandomForest.

4 Discussion

We have now accumulated considerable experience using NetWorkSpace, and while we have been generally pleased with ease of use and stability (the NetWorkSpace server routinely stays up for months), a few short comings in the design have become evident: (i) variable and workspace lifetime management are a little too subtle, (ii) an API for instantiating shared binding servers "on the fly" is needed, and (iii) an access control mechanism would be useful. The first tends to give rise to problems (e.g., unexpected re-incarnation of a variable) when variables and workspaces are being deleted for reasons other than to clean up while shutting down (as we mentioned, such deletions are sometimes used, for example, for out of band signaling). This is a fairly advanced usage and so not of major concern to most users, but it is something that we would like to rethink at some point. The second reflects a more common usage: a user wants a server to be dedicated to a particular instance of an application. This may be due to modularity considerations, performance concerns, or to simplify monitoring and control of the application. We have implemented a first cut at a "personal network-space server" to provide this ability. The third reflects the need to reduce the potential for accidental (or even malicious) cross-user interference. Twisted provides facilities that can help here, but perhaps the biggest challenge will be designing access controls that provide adequate protection but that do not overly complicate the user-level API or the server architecture and that do not adversely impact the inherent flexibility and heterogeneity of the current implementation.

Longer term plans include support for multi-threaded clients (via sharing of work-space objects, rather than having each thread instantiate a workspace object—this can be done now) and the addition of some sort of transaction capability (we are currently experimenting with a form of STM [15]).

Acknowledgements We gratefully acknowledge the Yale University Biomedical High Performance Computing Center and NIH Grant: RR19895, which funded the instrumentation used for portions of this work.

References

1. Message Passing Interface Forum.: MPI: A Message-Passing Interface Standard [Online]. Available: http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html (1995)
2. Gelernter, D.H.: Generative communication in Linda. ACM Trans. Program. Lang. Syst. **8**(1), 80–112 (1985)
3. Gelernter, D.H., Carriero, N.J.: Coordination languages and their significance. Comm. ACM **32**(2), 97–107 (1992)
4. Tierney, L., Rossini, A., Li, N., Sevcikova, H.: Snow: simple network of workstations [Online R Package]. Available: http://cran.r-project.org/web/packages/snow
5. Gelernter, D.H., Jagannathan, S., London, T.: Environments as First-Class Objects. In: ACM Conference on Principles of Programming Languages, pp. 98–110 (1987)
6. Gelernter, D.H., Jagannathan, S.: Programming Linguistics. MIT Press (1990)
7. Twisted Matrix Labs.: Twisted [Online]. Available: http://twistedmatrix.com/trac/wiki/Downloads (2008)
8. Lloyd, S.P.: Least squares quantization in pcm. IEEE Trans. Inf. Theory **28**(2), 129–136 (1982)
9. K-means Algorithm [Online]. Available: http://en.wikipedia.org/wiki/K-means
10. Hornik, K.: R FAQ [Online]. Available: http://cran.r-project.org/doc/FAQ/R-FAQ.html (2008)
11. Dhillon, I.S., Modha, D.S.: Concept decompositions for large sparse text data using clustering. Mach. Learn. **42**(1), 143–175 (2001)
12. Kuhn, M.: Caret: classification and regression training [Online R Package]. Available: http://cran.r-project.org/web/packages/caret (Mar. 2008)
13. Hastie, T., Tibshirani, R., Friedmann, J.: The Elements of Statistical Learning: Data Mining, Inference, and Prediction. Springer (2001)
14. Breiman, L.: Random forests. Mach. Learn. **45**(1), 5–32 (2001)
15. Jones, S.P.: Beautiful concurrency. In: Oram, A., Wilson, G. (eds.) Beautiful Code. O'Reilly (2007)