Making Infernal Grids Usable

C H Forsyth

Vita Nuova
3 Innovation Close
York Science Park
York YO10 5ZF

6 January 2006
`forsyth@vitanuova.com`

ABSTRACT

We regard a 'grid' as a specialised application of distributed systems. We describe the construction of several production grids implemented using the Inferno distributed operating system. The underlying technology was originally a convenient means to an end: both systems were developed as commercial projects, designed in close cooperation with the piper-paying end users, and that contributed more than the technology to the results being regarded as eminently usable by them. The technology did, however, help to speed development, and makes even its low-level interfaces easier to document, understand, and maintain. It represents all grid functionality, including the scheduler, as directory hierarchies, accessed through normal file IO operations, and imported and exported through the Internet as required.

1. Introduction

Vita Nuova applies novel technology to the construction of distributed systems. The background of the founders includes the successful development during the 1990s of systems for real-time auctions (of pigs and cars, separately) and on-line Internet game playing. These were significant distributed systems produced on commercial terms using conventional technology (C and Unix) in unconventional ways. For instance, the interfaces between the various components of the auction system were specified in the Promela protocol specification language, verified automatically using the SPIN protocol verifier, and the implementation as distributed communicating processes was derived directly from them. We later formed Vita Nuova to develop and exploit some (then) new systems originally developed by the Bell Labs research centre that produced Unix thirty years ago, Plan 9 and Inferno. Unlike Unix (and Unix clones) they were designed from the start with a networked world in mind.

In the past few years, Vita Nuova has undertaken several successful projects to build both resource-sharing and computational grids, for end-users. The aim was not to produce a toolkit, but usable deployed systems: 'end users' in our case meant the researchers and scientists, not users of a programming interface. Nevertheless, some scientists need a programming interface, and we want that to be straightforward. Various aspects of the underlying technology can help.

I shall describe three systems we built and the implementation process, and draw some conclusions. All three systems were implemented using the Inferno system. We do feel that its technological model made the systems easier to design and build, quickly, to good standards, and I shall suggest some reasons why. More important though is a key aspect of the development process: in each case we worked closely with the customers — the scholars and scientists that were the end users — to create a system that met their needs, that they could deem 'usable'.

2. Inferno

Inferno[1,2] is an operating system that provides an alternative universe for building distributed systems. The environment presented to applications is adopted from Plan 9,[3] and unorthodox. All resources are represented in a hierarchical name space that can be assembled dynamically to per-process granularity.[4] Resources include conventional file systems, but also devices, network interfaces, protocols, windowing graphics, and system services such as resource discovery and name resolution. Inferno runs 'native' on bare hardware but unusually, it can also run 'hosted' as an application under another operating system, including Windows and Unix-like systems. It provides a virtual operating system for distributed applications, including a virtual network, rather than acting as conventional 'middleware'.

Inferno applications access all types of resources using the file I/O operations: open, read, write and close. An application providing a service to be shared amongst many clients acts as a file server. A *file server* in this context is simply a program that interprets a common network-independent file service protocol, called Styx™ in Inferno and 9P2000 in Plan 9.[2,5] Distribution is achieved transparently by importing and exporting resources from one place to another using that protocol, across any suitable transport. An application runs in a local name space that contains the resources it needs, imported as required, unaware of their location. Resources can readily be securely imported across administrative boundaries. When running hosted, Inferno provides the same name space representation of resources of the host system (including devices and network interfaces) as it would provide running native. Thus for example, one can import the network interfaces of a Linux box into an application on Windows, to use Linux as a gateway. In all cases, Inferno presents the same narrow system call interface to its applications.

The Styx protocol itself is language and system independent. It has a dozen operations, mainly the obvious open, read, write and close, augmented by operations to navigate and operate on the name space and its metadata. It has been implemented on small devices, such as a Lego™ programmable brick,[6] has C, Java and Python implementations amongst others. A Styx server implementation in FPGA is being developed to support pervasive computing. It has been proposed for use in wireless sensor networks.[7] Blower et al.[8] provide Grid Services including work flow using their Java Styx implementation. The **v9fs** module of the Linux 2.6 kernel series provides direct access to 9P2000 and thus Styx through normal Linux system calls.[9]

3. Infernal grids

We applied Inferno and Styx to grid building. Perhaps one impetus was reading papers reporting research into things we had been doing quite casually with Plan 9 and Inferno for years, such as running interactive pipelines with components securely crossing authentication and administration with secure access through the Internet to the resources of many systems at once. I shall describe three: a demonstrator, which is still of interest for the range of resources it accessed; a data-sharing grid, which creates a secure logical data store from a dispersed collection; and a more traditional computational grid, which spreads a computational load across many computing elements, Condor-like.[10] Despite their different tasks, they are all built using the uniform Inferno mechanisms described above: all the resources are represented by Styx file servers, and can be accessed by traditional commands and system calls.

3.1. The demonstrator

The first system was a small experiment in building a simple 'grid' (at least in our own terms), that would later act as a demonstrator, for instance at Grid shows and when touting for business.[11] We had a collection of resources on different machines running different operating systems, and aimed to give access to them to anyone who connected to the system. 'Resources' included data, devices, and services, all connected and accessible remotely (securely). We wanted a good variety to emphasise that such a range could be handled by Inferno's single resource representation, and thus distributed. Amongst other things, we offered the following:

- a Lego™ Mindstorm™ programmable brick, built into a working clock[6]

- a Kodak™ digital camera under computer control
- an ODBC™ database
- collaborative activities: a shared whiteboard and multi-player card games
- a distributed computation service for image processing ('cpu service')
- a resource registry

It took three people about four weeks to put the system together, including writing the cpu service, the resource registry, the image processing demos, and the user interfaces. With some experience, it was later revised and refined, and some manual pages written, but retained the same implementation structure. Also deployed as part of the demonstration was an Internet Explorer plug-in we had written much earlier, that embedded the Inferno operating system environment on a web page, allowing web browsers to run Inferno to access our little demonstration 'grid' without having to install special software explicitly, and without our having to compromise our demonstration by using an HTTP gateway. A PC running native Inferno ran the gateway to the set of services, which were behind our firewall. Inferno used its public-key system to authenticate callers, who could then add their own machines to the pool of cpu servers in the registry to increase the processing power of the demo 'grid'.

The system was demonstrated at several conferences, including a UK academic Grid conference in June 2003.[11] The system was not intended for production use; it was not sufficiently fault-tolerant, for example. Even so, it allowed us to experiment with different ways of representing computational resources. It also showed that a realistic range of resources could be collected and accessed using Inferno's single method for resource representation and access: all the resources and services listed above were represented by file servers, and accessed by clients using open, read, write and close. The resources themselves were provided by different machines running different native operating systems, but all of them were also running Inferno.

3.2. Data grid for the humanities

The Center for Electronic Texts in the Humanities at Rutgers wished to use a distributed computing system to allow disparate collections of literary resources to be represented and accessed as a uniform whole, with the collection spanning different machines, platforms and administrations. The aim was to allow secure ''interactive collaboration toward common goals as if the users were working on a single large virtual computer''.[12] Tasks to be supported included not just viewing existing archives, but managing and modifying them, and adding new material, including annotations and links.

We were approached by one of the principals and, given the non-profit nature of the organisations involved, agreed to create the system 'at cost'. Some preliminary requirements capture and specification was done using e-mail and telephone before one member of staff was despatched to Rutgers to do the work. He spent a few days on site, developed the final specification in discussions with the end-users, implemented the supporting Inferno applications, refined it, installed the system on the relevant machines, and provided some training. The application is still in use today.

The implementation is straightforward: one or more archive browser applications run on a client, each building a name space suitable for the given task, incorporating name spaces from other servers or clients as needed to support the current collaboration. Since all resources look like files, the archive browser looks like a file browser, even though no physical file hierarchy corresponds to the view presented. It is important that machines are easy to add, that access is secure, and that a collaborator sees only the files that are part of the collaboration. Furthermore, the applications that use the resources are not available in source, and must remain unchanged.

3.3. Computational grid

The computational grid is perhaps more interesting, because it required significant new components. It was a full commercial development, and much larger scale. Even so, the same approach was followed: requirements capture, system specification, implementation, deployment, testing, and refinement.

The customer made extensive use of an existing set of scientific applications. They used them for their own R&D, but also to provide an analysis service to other companies. The application could consume as much computing power as it was offered. Fortunately, as often happens, the input data could be split into smaller chunks, processed independently, and those separate results collated or aggregated to yield the final result. Across its various departments, the company had many hundreds of Windows PCs, and a handful of servers, and they wished to use the spare cycles on the PCs to build a big virtual computational system.

Several members of Vita Nuova's development staff, including a project manager, worked with knowledgeable staff of the customer to refine and expand their initial set of requirements. They had previous experience with grid systems, and their initial requirements were perhaps therefore more precise than might otherwise have been the case.

Specification and implementation was actually in two stages, with minuted meetings and summary reports throughout. First, a proof-of-concept system was agreed, and produced by modifying the original demonstration software. We demonstrated to customer scientists (and management) that we could run existing Windows applications on a small collection of machines in our office, and that the performance improvement claimed could be achieved. We could then discuss the requirements and specification of the real system in detail, decide deliverable items, design an implementation and carry that out.

Of course, had the small demonstrator failed in fundamental ways, the project could easily have ended then, at small cost to the customer. In our experience, that is the usual approach for commercial projects. Of course, it does not account for many uncertainties: for instance, would the 'production' system scale up to many hundreds of nodes? It is not unusual for money to be kept back until the customer has received and accepted the deliverables of the contract. (I belabour these points to emphasise that there is considerable pressure on the software team to deliver something the customer does indeed find usable.)

The resulting system, now called 'Owen', has a simple model, that of a labour exchange,[13] forming the heart of a virtual time-sharing system. Worker nodes with time available present relevant attributes and qualifications to a scheduler at the exchange, which offers work from its queue. If a task is suitable, a worker does it, and submits the results, which are checked at the exchange (eg, for completeness or validity) before the corresponding task is finally marked 'done'. The scheduler accounts for faulty nodes and tasks. The largest unit of work submission and management is a *job*, divided into one or more atomic *tasks*. Each job has an associated *task generator* that splits the work into tasks based on some specified criteria peculiar to a given application. Usually the task split is determined by properties of the input data, and known before the job starts, but tasks can be created dynamically by the task generator if required. Workers request work as they fall idle, much as processors collect work on shared multiprocessor, rather than work being pushed to them in response to an estimated load.

The programming team had three people: one for the scheduler, one for the worker, and one to do the various graphical interface programs for grid control. It took six to eight weeks to produce the final deliverable, and install and test it. Subsequently it has undergone further development, to add capabilities not relevant to this discussion.

4. User and programmer interfaces

File-based representation as the lowest visible access level somehow gives access to the resources a solid 'feel'. It is easy to build different types of interface on that basis. I shall use the computational grid as an example here.

4.1. Node and job administration

Owen is administered through two simple 'point and click' applications. The Node Monitor, shown in Figure 1, shows the status of the computers available (or not) to do computation. Nodes can disconnect after accepting work, and reconnect much later to submit results. The node monitor distinguishes those cases. It allows nodes to be added and removed from the working set, suspended, and organised into named groups that are restricted to certain jobs, job classes or job owners.

The Job Monitor, shown in Figure 2, shows the status of jobs submitted by a user, including the progress made thus far. Jobs can be created and deleted, paused and continued, and their priority adjusted. Both graphical applications aim for a simple, uncluttered appearance, reflecting a simple model that hides the more complex mechanics of the underlying system.

4.2. Fundamental interface to the labour exchange

To make this more concrete, and perhaps help justify our choice of Inferno, let us look behind the graphical user interface. The starting point for design was the definition of a name space representation for computational resources. The Exchange's scheduler serves that name space. Each worker node attaches it to its own local name space, in a conventional place. We can look at it using the following commands:

```
mount $scheduler /mnt/sched
cd /mnt/sched; ls -l
```

The scheduler serves a name hierarchy, with the following names at its root, shown here as the result of the Unix-like `ls -l` above:

```
d-r-xr-x--- M 8  admin  admin 0 Apr 13 19:58 admin
--rw-rw-rw- M 8 worker worker 0 Apr 13 19:58 attrs
--rw-rw-rw- M 8  admin  admin 0 Apr 13 19:58 nodename
--rw-rw-rw- M 8 worker worker 0 Apr 13 19:58 reconnect
--r--r--r-- M 8 worker worker 0 Apr 13 19:58 stoptask
--rw-rw-rw- M 8 worker worker 0 Apr 13 19:58 task
```

Note that although each worker sees the same names, from the scheduler's point of view, each worker can be distinguished and given data appropriate to it. The files shown as owned by `worker` are those the worker can access. The worker can write to the `attrs` files to describe acceptable work and the properties of its node. A worker reads the `task` file to obtain offers of work; the read blocks until work is available. (In general, that is how publish/subscribe is done in a Styx environment: rather than having a special mechanism and terminology, an application simply opens a file to 'subscribe' to that data source on a server, and subsequent reads return data representing the requested 'events', as they become available, just as it does for keyboard, mouse or other devices.) Subsequent reads and writes exchange task-specific data with the job's task generator running as a process in the scheduler. That is processed by a task-specific component running in the worker. Amongst other things, the client-side component might receive a set of task parameters, and small amounts of data or a list of resources elsewhere on the network for it to put in the task's local name space for the application's use. The first task generators were specific to programs such as GOLD or CHARMM, so that customers could do productive work immediately. We then abstracted away from them. More recent generators support various common classes of computation (including those applications); the latest generator offers a simple job description language.

Job control and monitoring is provided by files in the `admin` directory:

```
d-r-xr-x--- M 4   rog admin 0 Apr 14 16:31 3
d-r-xr-x--- M 4   rog admin 0 Apr 14 16:31 4
d-r-xr-x--- M 4   rog admin 0 Apr 14 16:31 7
--rw-rw---- M 4 admin admin 0 Apr 14 16:31 clone
---w--w---- M 4 admin admin 0 Apr 14 16:31 ctl
--r--r----- M 4 admin admin 0 Apr 14 16:31 formats
--r--r----- M 4 admin admin 0 Apr 14 16:31 group
--r--r----- M 4 admin admin 0 Apr 14 16:31 jobs
--r--r----- M 4 admin admin 0 Apr 14 16:31 nodes
d-rwxrwx--- M 4 admin admin 0 Apr 14 16:31 times
```

Access is restricted to the `admin` group. Each job has an associated subdirectory (3, 4 and 7 here) containing job-specific data:

```
--rw-rw---- M 4 rog admin 0 Apr 14 17:08 ctl
--rw-rw---- M 4 rog admin 0 Apr 14 17:08 data
--rw-rw---- M 4 rog admin 0 Apr 14 17:08 description
--r--r----- M 4 rog admin 0 Apr 14 17:08 duration
--r--r----- M 4 rog admin 0 Apr 14 17:08 group
--r--r----- M 4 rog admin 0 Apr 14 17:08 id
--r--r----- M 4 rog admin 0 Apr 14 17:08 monitor
```

To create a new job, a program opens the `admin/clone` file, which allocates a new directory labelled with a new job number, and yields a file descriptor open on its `ctl` file. Control requests are plain text. For example,

`load` *jobtype arg ...*

selects the task generator for the job and provides static parameters for the job, `start` sets it going, `stop` suspends it, and `delete` destroys it. Reading `id` returns the job's ID. Reading `duration` gives a number representing the lifetime of the job. Reading `group` returns the name of the group of worker nodes that can run the job. The first read of the `monitor` file returns the job's current status, and each subsequent read blocks until the status changes, following the same idiom as for the `task` file above. Note that only the grid administrator and the job owner have permission to access these files. Files in the `admin` directory itself describe or control the whole grid.

4.3. Levels of interface

The Job and Node monitors are graphical applications that present a conventional graphical interface, and they are the ones used by most users. The programs act, however, by opening, reading and writing files in the name space above. Thus the node monitor reads a file to find the current system status, which it displays in a frame. The job monitor kills a job by opening the appropriate control file (if permitted) and writing a `delete` message into it. This makes them easy to write and test. The worker node's software also acts by opening, reading and writing files.

Given the description of the scheduler's name space, and the protocol for using each of its files, the three software components (monitors, worker and scheduler) can be written and tested independently of each other. It is easy to produce either 'real' files or simple synthetic file servers that mimic the content of the real scheduler's files for testing. Conversely, when testing the scheduler, one uses the Inferno shell to operate on its name space and script test cases. This encourages independent development.

At a level below the graphical interace, the system provides a set of shell functions to be used interactively or in scripts. Few end-users currently work at this level, but it is discussed briefly here to suggest that it is not too demanding. Here is a script to submit a simple job to the grid:

```
run /lib/sh/sched    # load the library
mountsched      # dial the exchange and put it in name space
start filesplit -l /tmp/mytasks test md5sum
```

The last command , `start`, is a shell function that starts a new job. In this case, a `filesplit` job, creating separate tasks for each line of the file `/tmp/mytasks`, each task doing an MD5 checksum of the corresponding data. The implementation of `start` itself is shown below:

```
fn start {
    id := ${job $*}
    ctl $id start
    echo $id
}
```

It calls two further functions `job` and `ctl`, that interact with the labour exchange's name space described earlier. `Job` clones a job directory and loads a task generator. Ignoring some error handling, it does the following:

```
subfn job {
    {
        id := ‘{cat}
        result=$id
        echo load $* >[1=0]
    } $* <> /mnt/sched/admin/clone   # open for read/write
}
```

and `ctl` sends a control message to a given job:

```
fn ctl {
    (id args) := $*
    echo $args > /mnt/sched/admin/$id/ctl
}
```

Several of the task generators also invoke application-specific shell functions on labour exchange and client, allowing tailoring to be done easily. For instance, on the client a function `runtask` is invoked to start a task running on a client, and `submit` yields the result data (if any) to return to the exchange. The `test` job used above is defined on the client as follows:

```
load std
fn runtask {
    $* > result # run arguments as command
}

fn submit {
    cat result
}
```

The `runtask` for a Windows application is usually more involved, constraining the code that runs, checking preconditions, and providing its data in the place and format it expects. Here is a version for an image-processing task:

```
fn runtask {
    # get files
    check gettar -v     # unpack standard input as tar file

    # run
    cd /grid/slave/work
    check mkdir Output
    check {
        os -n $emuroot^/grid/slave/image/process.exe <image >Output/image
    }
}
fn submit {
    check puttar Output
}
```

A corresponding job class specification sits on the exchange. It too is a set of shell functions: `mkjob` to prepare global parameters for a job; `mktask` to prepare task-specific parameters; `runtask` to provide a task's data and parameters to a worker; `endtask` to check its results; and `failedtask` to do error recovery. Such specifications range from 70 to 140 lines of text, the largest function being 10 to 20 lines.

The task generators mentioned earlier are modules in the Limbo programming language. They range between 200 and 700 lines of code, and require knowledge of interfaces shared with the exchange. In practice, now that we have defined general-purpose generators that invoke shell scripts or interpret our job description notation, it is unlikely that anyone else will ever write one, but the interface is there to do it.

4.4. Job description

One task generator interprets a simple form of job description. It specifies a job and its component tasks declaratively, in terms of its file and value inputs. To avoid fussing with syntax, we use S-expressions. For example, the following trivial job calculates the MD5 checksum of each line of the file `myfile`, using a separate task for each line:

```
(job
   (file (path myfile) (split lines))
   (task exec md5sum))
```

The little declarative language allows specification of a set of tasks as the cross-product of static parameters and input files or directories, and has directives to control disposition of the output. It can optionally specify shell functions to invoke at various stages of a job or task.

5. Related work

The Newcastle Connection[14] made a large collection of separate UNIX systems act as a single system (in 1981!), by distributing the UNIX system call layer using either modified kernels or libraries. It was easy to learn and use because the existing UNIX naming conventions, and all the program and shell interfaces familiar from the local system, worked unchanged in the larger system, but still gave access to the expanded range of systems and devices in the network. (Their paper mentions earlier systems that also extended UNIX in various ways to build distributed systems from smaller components.) The deployment and use of their system was, however, complicated by limitations built in to the 1970s UNIX design, notably that device control and user identification had been designed for a self-contained system, and perhaps more important, services provided outside the kernel had no common representation, and were not distributed by the Connection. Plan 9 from Bell Labs[15] was by contrast were designed for distribution from the start, and takes a radical approach to resolve the problems with UNIX: in particular, it represents all resources in a uniform way, regardless of origin, and can use a single mechanism to distribute everything. By virtualising Plan 9, Inferno spreads those benefits throughout a heterogenous networks. The underlying protocol itself can, however, be used independently.

Minnich[16] has developed an experimental system, **xcpu**, for building clusters of compute nodes using the **v9fs** Linux kernel module. As usual with 9P2000 or Styx, the lowest-level interface to the system is a hierarchy of names, representing nodes and processes on a set of heterogeneous cpu servers. Instead of submitting a job to a scheduler, as in an old-style ‘batch’ system, **xcpu**’s model is that of an interactive time-sharing system. A user requests a set of nodes of particular type (eg, compute or IO) and imports a name space from each one, which he attaches to his local Linux name space. Files in each node’s name space represent attributes of the node (such as architecture and ability), the program to run, and its standard input and output. Now the user can use any desired mixture of ordinary user-level Linux system calls, commands and scripts to interact directly with any or all of those nodes, seeing results as they appear. For example, shell commands could set up a group of nodes to run an MPI application, copy the executable to each node in parallel, and then start them, along the following lines:

```
for i in $NODES; do
    cp my-mpiapp-binary /mnt/9/$i/exec &
done
wait    # for all copies to finish
for i in $NODES; do
    echo mpiapp $i $NODES >/mnt/9/$i/argv &&
        echo exec >/mnt/9/$i/ctl
done
```

Output collection, error detection, and conditionals use the command language and commands already familiar to the users from daily use off-grid. Minnich observes that of course higher-level interfaces are possible and often desirable; note, however, that this is the lowest-level interface, and it can be used directly without much training.

Legion[17] used a virtual operating system model to build a distributed object system, specifically for grid applications. The naming hierarchy locates objects, which have object-specific operations, and objects communicate (below the name space) using messages carrying method call parameters and results. The system provides support for object persistence, fault-tolerance, scheduling, and more, but the programming and user interface is unique to it, and non-trivial. Grimshaw et. al[18] compare Legion and Globus GT2 at many levels, including design principles, architecture, and the implications of the passive object model then used by Globus to Legion's active object model. Differences are, indeed, legion. Most important here, though, is the observation that ''it was very difficult [with Legion] to deliver only pieces of a grid solution to the [grid] community, so we could not provide immediate usefulness without the entire product'' but Globus (by providing short-term solutions to key problems) ''provided immediate usefulness to the grid community''. Architecturally, they contrast a bottom-up approach used by Globus to the top-down design of Legion. They describe the then emerging Open Grid Services Architecture as a collection of ''specialized service standards that, together, will realize a metaoperating system environment'', similar in aim therefore to Legion, with a different RPC mechanism based on SOAP, WSDL, etc. Given the relative complexity of Legion itself, though, and the authors' admission that it was most useful as an 'entire product', that could be double-edged.

There are several commercial suppliers of grids, sometimes offshoots of university experience.[17,19] Hume describes specialised software for fault-tolerant data processing clusters,[20,21] processing huge telephony billing data sets to deadlines. Google has produced grid systems that are impressive in scale and resiliency,[22] based on a general underlying storage subsystem,[23] with a specialised high-level language to ease programming.[24] These developers have enthusiastic end-users, and argue that their systems (particularly Google's) are objectively easier to use and more productive than ones they replaced, based on metrics such as size and complexity of programs, and comparative system performance.

6. Discussion

Beckles[25] lists some properties by which 'usable' grid software might be assessed: ''engagement with the intended user community; APIs and other programmatic interfaces; user interfaces; security; documentation; and deployment issues''.

Our Owen computational grid software was developed in close consultation with its initial users, but has subsequently been provided to other customers. They have similar application profiles to the original users; all of them continue to do useful work with it. Some are contented users of newer facilities such as its job specification notation. We have also made the software available to a few universities and research organisations. Results there are mixed, limited mainly by documentation. If they wish to use it as originally designed, it works well. Others quite reasonably need much more flexibility, and although that is often supported by the underlying software, the extra documentation to allow them to use it easily is not yet available.

At a lower level, however, our systems are undoubtedly distinctive in having clients access and control the grid using file-oriented operations. Consequently, the specification of the programming interface — such as the manual page for the scheduler — defines the hierarchy of names it serves, and the contents of the files therein, including a list of control files, the textual messages that can be written to them, and their effect. What are the advantages of this representation? It is directly accessible through any programming language in which there are constructions (or libraries) providing explicit file access with open, read and write. Even better, in some environments it can be accessed directly by existing commands. As discussed above, a small file of shell functions allow grid jobs to be created and controlled from the Inferno command line as discussed above. That is currently of little interest to many end-users, who largely wish the whole process to be invisible: to have the computational distribution happen without their explicit intervention. Some, however, such as those who already write their own Perl scripts, are able, with only a little training, to provide their own interfaces to the grid. Beckles argues for APIs that support 'progressive disclosure' of functionality, and we find the underlying file-based metaphor well supports that. The demonstrator grid mentioned above showed that the metaphor can be applied to a respectable range of resource types; there are many other examples to show it is not limiting.[26, 3, 5]

Another possible difference compared to a Globus[27] or Web Services[28] interface is that instead of learning both protocol and mechanism, with the file-oriented interface, such users must learn only the names and the protocol: the file-access mechanism is almost always quite familiar to them, and they can carry across their intuitions about it. As it happens, the protocol for the use of a given name space is also usually simpler and more straightforward to use than language-specific APIs full of complex data structures or objects. Indeed, the implementation of file servers has not been discussed here, but it is notable that although the Styx protocol has messages of a specific structure, there are only 13 operations, most of them obvious (eg, open, close, read, write) with obvious parameters, and every file server implements the same operations. Compared (say) to plug-ins to a web server, there are neither programming interfaces nor data structures imposed on the servers by their surroundings; they can manage their own internals as they like, as self-contained programs. For instance, some are concurrent programs, but many are simple programs that run in a single process. Within Inferno, there is a system primitive `file2chan` that makes it easy to write applications that serve only a handful of names. Lacking (and a serious omission) are good tutorials on writing 'full' file servers from scratch.

For wide-scale release, it is easy to underestimate the effort required to document both infrastructure and applications, especially to suit a range of end-users (ie, scientists and programmers). In 1999-2000, for the first public release of Inferno itself, a team of five spent many months preparing its documentation and assembling the distribution (and Inferno is not a huge system). That documentation was intended for programmers (perhaps even system programmers). It can be harder still to prepare things for end users, even when the applications have been kept simple. We do not yet consider our grid documentation to be up to our standards.

We originally distributed the client software from a CD using the same mechanism as Inferno, but that was clumsy. The grid client software is now typically installed simply by copying a small hierarchy (a few megabytes), for instance from a USB memory stick or a shared file store.

Conclusions

Supposing that within the constraints of our time and treasure, we have indeed managed to build grids that customers found usable, and with underlying technology and interfaces that we consider usable, are there any general lessons? They are mainly traditional ones. As implied in the company history in the Introduction, we have successfully applied them in the past, outside Inferno.

We had a clear and indeed limited design (''do one thing well''). Rather than provide an abstract interface to a big family of possibilities, as is done by some of the Web Service specifications, we decided on one that was 'fit for purpose'. We used mature technology with which we had much design and implementation experience; its definition is compact and stable. The system should help separate concerns and provide a focus for the design. In our case, system-level mechanisms do distribution and station-to-station authentication. The name space provides the focus of discussion during initial design meetings, and its description acts as a high-level contract between components. Minimisation of mechanism is a good engineering principle; we were working with an infrastructure that provides a narrow interface. On the other hand, if the system-provided mechanisms are too primitive (''message passing'' or ''remote procedure call'') then there is little structural support for application development, no common currency, and little chance of either users or programmers being able to re-use knowledge. Using simple metaphors that make sense to them must surely help, especially if the just one can be used throughout.

Most important, as Beckles suggests,[25] interaction with end users is essential. Our various projects have been successful, and the resulting new systems worked well for their users, partly because the underlying technology worked well and was easy to use, but also because we dealt with them as normal commercial projects, working closely with customers to provide something that suited them. As important as that interaction is an imposed discipline. We have fortunately been able to deal directly with end users who really must get their work done; the amazing politics now associated with the e-science grid world is irrelevant to them and to us. Furthermore, commercial requirements—certainly for a small company—mean that the result must meet deadlines, satisfy customer expectations, and keep to budget. We find this helps to focus mind and energy wonderfully.

Acknowledgements

Roger Peppé and Chris Locke designed and implemented the labour exchange; Peppé subsequently added the job description language and other improvements, and wrote the documentation. Danny Byrne wrote the graphical interface applications.

References

1. Sean Dorward, Rob Pike, David L Presotto, Dennis M Ritchie, Howard Trickey, Phil Winterbottom, ''The Inferno Operating System,'' *Bell Labs Technical Journal* **2**(1), pp. 5-18 (Winter 1997).
2. *Inferno Programmer's Manual, Third Edition,* Vita Nuova Limited (2000).
3. Rob Pike, Dave Presotto, Ken Thompson, Howard Trickey, Phil Winterbottom, ''The Use of Name Spaces in Plan 9,'' *Proceedings of the 5th ACM SIGOPS European Workshop*, Mont Saint-Michel (1992).
4. Rob Pike, Dennis M Ritchie, ''The Styx Architecture for Distributed Systems,'' *Bell Labs Technical Journal* **4**(2), pp. 146-152 (April-June 1999).
5. C H Forsyth, ''The Ubiquitous File Server in Plan 9,'' *Proceedings of the Libre Software Meeting*, Dijon, France, Vita Nuova Limited (5-9 July 2005).
6. Chris Locke, *Styx-on-a-Brick,* Vita Nuova Limited, York, England (June 2000).
7. Sameer Tilak, Bhanu Pisupati, Kenneth Chiu, Geoffrey Brown, Nael Abu-Ghazaleh, ''A File System Abstraction for Sense and Respond Systems,'' *Workshop on End-to-End, Sense-and-Respond Systems, Applications and Services*, Seattle WA, pp. 1-6 (June 2005).
8. Jon Blower, Keith Haines, Ed Llewellin, *Data streaming, workflow and firewall-friendly Grid Services with Styx,* e-Science Centre, University of Reading (2005).
9. Eric Van Hensbergen, Ron Minnich, ''Grave Robbers from Outer Space: Using 9P2000 Under Linux,'' *Proceedings of the USENIX 2005 Annual Technical Conference, FREENIX Track*, Anaheim, CA, pp. 83-94 (April 2005).
10. M Litzkow, M Livny, M W Mutka, ''Condor - a Hunter of Idle Workstations,'' *Proceedings of the 8th International Conference of Distributed Computing Systems*, pp. 104-111 (June 1988).
11. Michael Jeffrey, *Inferno Grid Applications,* Vita Nuova Limited, York, England (June 2003).
12. Brian Hancock, ''A Brief Introduction to the Humanities Grid,'' *Library Hi Tech News*, New Brunswick, New Jersey(8), pp. 32-33, Rutgers University Libraries (2004).

13. Roger Peppé, *Owen - A Labour Exchange for Computational Grids,* Vita Nuova Limited, York, England (April 2004).

14. D R Brownbridge, L F Marshall, B Randell, ''The Newcastle Connection or UNIXes of the World Unite!,'' *Software Practice and Experience* **12**(6), pp. 1147-1162 (1982).

15. Rob Pike, Dave Presotto, Sean Dorward, Bob Flandrena, Ken Thompson, Howard Trickey, Phil Winterbottom, ''Plan 9 from Bell Labs,'' *Computing Systems* **8**(3), pp. 221-254 (Summer 1995).

16. Ron Minnich, ''XCPU operation,'' LA-UR-05-7562, Los Alamos National Labs (July 2005).

17. Andrew S Grimshaw, Anh Nguyen-Tuong, William A Wulf, ''Campus-wide Computing: Early Results Using Legion at the University of Virginia,'' CS-95-19, University of Virginia (March 1995).

18. A S Grimshaw, M A Humphrey, A Natrajan, ''A philosophical and technical comparison of Legion and Globus,'' *IBM Journal of Reesearch and Development* **48**(2) (2004).

19. S Zhou, J Wang, X Zheng, P Delisle, ''Utopia: A load sharing facility for large, heterogeneous distributed computing systems,'' *Software — Practice and Experience* **23**(12), pp. 1305-1336 (December 1993).

20. Andrew Hume, Scott Daniels, Angus MacLellan, ''Ningaui: A Linux Cluster for Business,'' *Proceeding of USENIX 2002 Annual Technical Conference (FREENIX track)*, Monterey, California, pp. 195-206 (June 10-11, 2000).

21. Andrew Hume, Scott Daniels, Angus MacLellan, ''Gecko: Tracking a Very Large Billing System,'' *Proceedings of 2000 USENIX Annual Technical Conference*, San Diego, California (June 18-23, 2000).

22. Jeffrey Dean, Sanjay Ghemawat, ''MapReduce: Simplified Data Processing on Large Clusters,'' *Proceedings of the Sixth Symposium on Operating System Design and Implementation*, San Francisco, California (December 2004).

23. Sanjay Ghemawat, Howard Gobioff, Shun-Tak Leung, ''The Google File System,'' *Proceedings of the Symposium on Operating System Principles*, Bolton Landing, New York (19-22 October 2003).

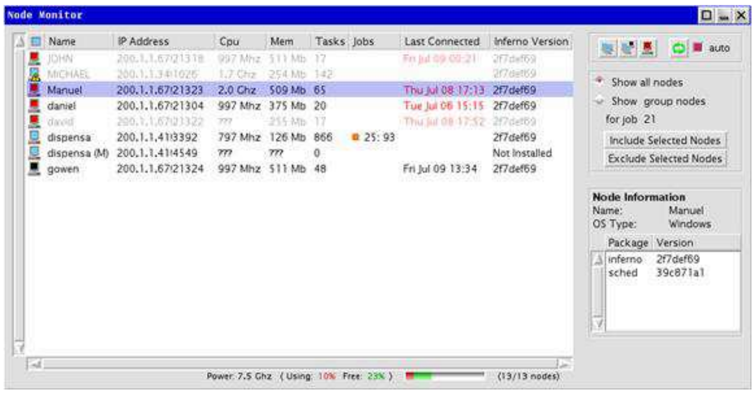
24. Rob Pike, Sean Dorward, Robert Griesemer, Sean Quinlan, *Interpreting the Data: Parallel Data Analysis with Sawzall (Draft),* Google, Inc (2005).

25. Bruce Beckles, *Re-factoring grid computing for usability,* University of Cambridge Computing Service, Cambridge, England (2005).

26. Dave Presotto, Phil Winterbottom, ''The Organization of Networks in Plan 9,'' *Proceedings of the Winter 1993 USENIX Conference*, San Diego, California, pp. 271-280 (1993).

27. I Foster, C Kesselman, ''Globus: A Metacomputing Infrastructure Toolkit,'' *Intl Journal of Supercomputer Applications* **11**(2), pp. 115-128 (1997).

28. David Booth, Hugo Haas, Francis McCabe, Eric Newcomer, Michael Champion, Chris Ferris, David Orchard, ''Web Services Architecture,'' Technical Report http://www.w3.org/ws-arch/, W3C (February 2004).



Node Monitor

Name	IP Address	Cpu	Mem	Tasks	Jobs	Last Connected	Inferno Version
JOHN	200.1.1.67!21318	997 Mhz	511 Mb	17		Fri Jul 09 00:21	2f7def69
MICHAEL	200.1.1.34!1026	1.7 Ghz	254 Mb	142			2f7def69
Manuel	200.1.1.67!21323	2.0 Ghz	509 Mb	65		Thu Jul 08 17:13	2f7def69
daniel	200.1.1.67!21304	997 Mhz	375 Mb	20		Tue Jul 06 15:15	2f7def69
david	200.1.1.67!21322	???	255 Mb	17		Thu Jul 08 17:52	2f7def69
dispensa	200.1.1.41!3392	797 Mhz	126 Mb	866	25: 93		2f7def69
dispensa (M)	200.1.1.41!4549	???	???	0			Not Installed
gowen	200.1.1.67!21324	997 Mhz	511 Mb	48		Fri Jul 09 13:34	2f7def69

Power: 7.5 Ghz (Using: 10% Free: 23%) (13/13 nodes)

auto

[✓ Show all nodes]
Show group nodes for job 21

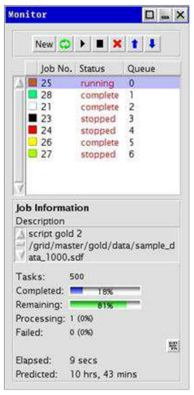
Include Selected Nodes
Exclude Selected Nodes

Node Information

Name: Manuel
OS Type: Windows

Package	Version
inferno	2f7def69
sched	39c871a1

Figure 1. The Node Monitor shows the status of computers forming the virtual computer



Monitor

New

	Job No.	Status	Queue
🟫	25	running	0
🟩	28	complete	1
⬜	21	complete	2
⬛	23	stopped	3
🟥	24	stopped	4
🟨	26	complete	5
🟩	27	stopped	6

Job Information

Description

script gold 2
/grid/master/gold/data/sample_d
ata_1000.sdf

Tasks: 500
Completed: 18%
Remaining: 81%
Processing: 1 (0%)
Failed: 0 (0%)

Elapsed: 9 secs
Predicted: 10 hrs, 43 mins

Figure 2. The Job Monitor shows job status and provides job control