RC24788 (W0904-091) April 21, 2009
Computer Science

IBM Research Report

Service Oriented File Systems

Eric Van Hensbergen, Noah Evans
IBM Research Division
Austin Research Laboratory
11501 Burnet Road
Austin, TX 78758

Phillip Stanley-Marbell
IBM Research GmbH
Zurich Research Laboratory
8803 Rüschlikon
Switzerland

Research Division
Almaden - Austin - Beijing - Cambridge - Haifa - India - T. J. Watson - Tokyo - Zurich

Service Oriented File Systems

Eric Van Hensbergen
IBM Research Austin
bergevan@us.ibm.com

Noah Evans
IBM Research Austin
npevans@us.ibm.com

Phillip Stanley-Marbell
IBM Research Zurich
pst@zurich.ibm.com

Abstract

Service Oriented Architectures (SOAs) are a loose coupling of network services providing methods for systems development and integration. Interoperability between different systems and programming languages is provided via communication protocols and well defined messages. The recent development trend has been to favor RESTful approaches for these interfaces, which encode relevant context and semantic metadata into the URL of an HTTP GET or PUT operation.

We observe that this approach is essentially a simplified web-instantiation of synthetic file system based service interfaces, such as those originally pioneered by UNIX and later the Plan 9 and Inferno operating systems. In this paper we advocate the collapse of the software stack by abstracting the underlying transport and naming details, and accessing RESTful services via standard file system interfaces. We explore the research challenges and opportunities presented by taking such an approach to building comprehensive dynamic distributed systems appropriate for large scale cloud computing.

1 Introduction

Service-oriented architectures (SOAs) are networked software infrastructures in which resources are made available through a transactional interface [19]. They typically comprise collections of reusable application modules, each of which expose standardized interfaces. The use of well-defined protocols and message formats to communicate between these services decouples application implementations from one another, providing greater degrees of flexibility and composability. The popularity of the web has lead to many SOA runtimes adopting HTTP encapsulated protocols such as SOAP [12].

The emergence of *Web 2.0* and *cloud computing* has extended the SOA development paradigm from business process services to complete computing environments, involving everything from the customer-facing interfaces to the back-end database. These new classes of application have evolved from client-server based traditional RPC exemplified by SOAP, to *representational state transfer (REST)* [6] mechanisms as their common communication architecture.

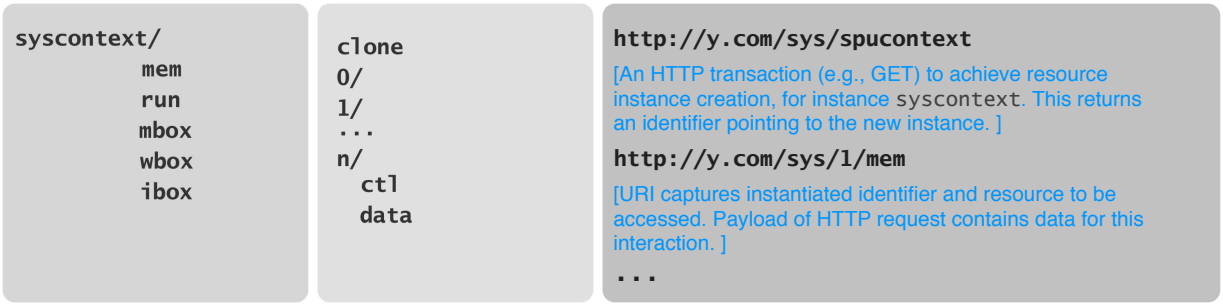
RESTful applications differ from their RPC counterparts in relying principally on *resource identifiers* versus *command methods.* In web environments, this results in operational semantics being encoded as part of the URL of HTTP requests, and relying on HTTP methods (GET, PUT, POST, DELETE) as the sole methods. This allows service components to access and navigate service infrastructures without requiring complete knowledge of the resource capabilities or data structures. Existing runtimes each use their own language-specific bindings for accessing and addressing these service methods.

It is our belief that it is the responsibility of systems software to provide a unified language- and network-neutral interface to its users. As such, we believe that the operating system should provide facilities for addressing and interaction of SOA interfaces. We feel that enabling interaction via system-provided interfaces enables a new degree of composability and flexibility in much the same way as pipes unlocked tool-based approaches in UNIX [10].

Our proposal is motivated by our observation that the semantically significant URL path component of RESTful operations is similar in many ways to synthetic file system environments such as UNIX's /proc, or those used in the Plan 9 or Inferno operating systems (Figure 1). The introduction of the *File Systems in User Space (FUSE)* and 9P file systems into the mainline Linux kernel has further enabled the use of such file system interfaces as an applications component in mainstream environments. We propose the use of such mechanisms to allow RESTful interfaces to be managed, organized, and interconnected by the operating system instead of purely by an SOA runtime. It is our belief that using synthetic file systems to establish operating system based uniform interface abstractions and application orchestration will further enable the benefits of SOAs while also enabling a seamless distributed computing environment across platforms to enable large-scale "cloud" environments.

2 Background

UNIX pioneered the concept of treating devices as files, providing a uniform interface to system hardware. In the 8th edition, this methodology was taken further through the introduction of the /proc synthetic file system to



(a). An example "Old Unix style" virtual filesystem, SYSFS; data exchange via files, but control and resource instance creation via `ioctl()` and `create()` system calls.

(b). A typical dynamic fileserver in Plan 9/Inferno; file abstractions are used for resource instance creation (`clone`), control (`ctl`) and data (`data`).

(c). An example RESTful interface to a system service exposed over HTTP.

Figure 1: Illustration of a service–oriented file system interface

manage user processes [8]. Synthetic file systems are comprised of elements with no physical storage, i.e., the files represented are not present on any disk. Instead, operations on the file communicate directly with the kernel sub-system or application providing the service. This methodology has persisted over the evolution of UNIX and UNIX-like systems, and the Linux kernel now supports several synthetic file systems representing devices, process control, as well as access to system services and data structures.

The Plan 9 [14] and Inferno [15] operating systems took the file system metaphor further, using file operations as the simple, well-defined interface to all system and application services. As such, interfaces to all kernel subsystems, from the networking stack to the graphics frame buffer, are represented within synthetic file systems. User-space applications and services may also export their own synthetic file systems, in much the same way as the kernel interfaces. Common services such as domain name service (DNS), authentication databases, and window management are all provided as file systems. Even end-user applications such as editors and e-mail systems export file system interfaces as a means for data exchange and control. The benefits and details of this approach are covered in detail elsewhere [16].

The intuition behind the design was based on the assumption that any programmer knows how to interact with files. Every programming language has the means of interacting with file systems, and network protocols (including HTTP) for remote access of file hierarchies are readily available. Application frameworks implemented as file systems eliminate many of the operating system and language portability problems that occur when implementers develop new protocols and interfaces—the responsibility for the interactions occur at the systems level, making a new API unnecessary. Programs just navigate a file hierarchy open the relevant files and read/write the necessary data to the proper location.

In addition to providing language portability, synthetic file systems provide data portability as well. They can act as intermediaries between the physical data and the presentation, interpreting data in the most appropriate way for a given application. Again this moves the responsibility for ensuring data from the user to system, removing much of the burden of data conversion from the user. For example files of one type, say xml data, can be presented as yaml and vice versa. Further examples of this process appear in [5].

Synthetic file systems have the unique property of being implementable either within the operating system or in user space, but still providing a single consistent interface through the operating system's file system name space. Using synthetic file systems as interfaces for software implementations decouples software functionality from decisions about implementation location, programming language bindings, or runtime system support.

3 Approach

Following Plan 9's example, we propose the pervasive use of *synthetic file systems* as a resource interface abstraction, and *an augmented form of UNIX pipelines* together with application of *dynamic private name spaces* to achieve this goal. We believe that by collapsing the infrastructure of service-oriented architecture stacks back into the system, we can return the flexibility and elegance of the composable modular approach exemplified by UNIX.

The presence of service interfaces within the operating system name space allows for natural naming and addressability, freeing services from location-dependent numeric identifiers such as IP addresses and port numbers. The use of simple, consistent distributed file sys-

tem protocols together with automated discovery and registry mechanisms can be used to bridge cluster resources into a comprehensive name space. Application of dynamic private name space semantics can be used to enforce context and security considerations.

The UNIX pipeline model [10] established a simple mechanism for composing collections of *programs* that read and write from their standard input/output streams, into *workflows* or *applications* of unlimited sophistication. We believe that a straightforward evolution of this same model can be used to construct dynamic collections of applications across heterogeneous networked computing platforms. This approach keeps modular components loosely coupled, maximizing their re-use and maintaining the principle of "do one thing well" that foundational UNIX applications were based upon. We believe such decoupling is vital to maximizing composability.

Presenting the interfaces through the systems software also decouples the addressing and network aspects of the service, allowing applications without HTTP components to interact with the services and data presented by SOA infrastructures. Such an approach doesn't prohibit HTTP based interaction with these resources, as the HTTP server can directly access the represented interface through the file system which should allow for an overall simplification to the HTTP server implementation.

3.1 Application: Interface

The pervasive appeal of the web has moved its function from simple query and information display to a full-blown applications environment. Web applications are becoming a dominant force driven by the desire to enable collaboration as well as location and platform independence. The current approach of many of these web applications involve JavaScript front-ends interacting with application server backends using *Asynchronous JavaScript and XML (AJAX)*. This decouples the user interface implementation from the back-end logic and storage management, allowing different interfaces to be built for different client platforms. The hierarchical nature of widget-based user interfaces maps easily to file-system-based structures, and, when combined with the concept of dynamic private name spaces, allows for "mashups" of interfaces from multiple back-end server implementations to be combined in a coherent visual interface.

The Octopus project [1] is an effort to look at synthetic file system based server implementations for such subdivided application environments. Octopus structures interfaces such that they can be expressed as a file hierarchy, and then uses *Op*, a simple CRUD protocol, for communication between user-facing and server-side components.

3.2 Application: Services

By utilizing synthetic file systems as the primary interface for application components, developers can both enable and take advantage of the composition of collections of applications hosted across a network. By decoupling service interface from runtime infrastructure, such systems increase modularity, and allow the developer to integrate possibly-different runtime systems, maximizing performance, efficiency and reliability by choosing the best tool for the job.

Libferris [9] is an example infrastructure which exposes file system interfaces to various data collections and services, such as XML, database queries, and so forth. This permits interesting composite applications to be constructed even from a command shell environment. Google's Chubby lock service [4] is another example of a synthetic file system providing a commonly required system component.

3.3 Application: Management

The provisioning and management of clusters can also be abstracted through synthetic file systems. The Xcpu infrastructure [11] provides a dynamic synthetic file system name space allowing process creation, management of, and interaction with created processes. A top-level provisioning control file is used to instantiate new processes. Subdirectories per process contain information analogous to the `proc` file system in UNIX, as well as control files which can be used to manipulate execution, and files representing standard input, standard output, and standard error. Computation servers can make this Xcpu synthetic file system accessible over a network, allowing remote systems to initiate and control execution on the server.

We are currently developing extensions to this methodology that allow dynamic instantiation of new logical partitions within cloud infrastructures, all controlled via synthetic file system interfaces. By layering machine provisioning as well as remote service execution, monitoring, and control through a cohesive name space, we can enable a seamless model of distributed computing without the complexity overhead of single-system image [2].

In order to orchestrate services instantiated in such a manner, we are exploring a distributed extension of the UNIX pipe abstraction which incorporates concepts of one-to-many distribution and subsequentcoalescingof results allowing for a natural instantiation of map/reduce style [7]. In support of this activity we are investigating support for passing references to synthetic file hierarchies as well as traditional character streams over these negotiated channels.

3.4 Application: Hybrid Systems

The technique of employing synthetic file systems as abstractions for resources can also be used to manage and interact with emerging *hybrid architectures*—multicore environments where cores have different architectural properties (e.g., different ISAs, or non-programmable hardware accelerators for tasks such as media encoding/decoding or encryption). Such architecturesrequire a looser coupling than typical SMP approaches to multicore systems. At the same time, the capabilities of such heterogeneous components goes well beyond the ability of typical device interfaces.

Spufs [3], a programming interface for the Cell processor, is one example of a synthetic file system interface to such hybrid architectures. In Spufs however, the exposed interface is not entirely through a synthetic file system, as several control interactions with the hardware can only be performed via `ioctl()` system calls. The approach we propose, is to enforce *all* configuration and data exchange through a synthetic file system, which would then naturally enable complete control of such hardware over a network, and in a distributed system.

We believe that deep synthetic hierarchies represent the right command and control interface for such heterogeneous environments. By incorporating management and control into the file system, these units can be controlled across the network and integrated into distributed computing schemes such as cloud computing.

4 Research Challenges

While synthetic file systems are relatively straightforward to interact with from the client, developing synthetic file servers has historically been considerably more difficult. Developers must not only know how to structure their service, they have to have an in-depth understanding of file system operations, semantics, and in some cases accounting for file system structures. Scalable multi-threaded and asynchronous synthetic file servers are even more challenging to implement.

RESTful approaches and synthetic file systems share common interface design pitfalls. It is often difficult for the developer to identify the right level of expressiveness to represent within the resource identifier (URL for REST, path name for file systems), and how much to encode within the content. For example, the elements of a data structure can be represented together in a single file using XML or s-expressions—or they can be spread amongst several files within the hierarchy allowing the user application or script to only get or put the granular element. Different circumstances require different decisions as to the granularity of the data represented in a file. We have also encountered situations in which both representations are desired.

Topology is another common design issue between file system and RESTful interfaces. There are cases in which the natural layout of the data isn't easily represented in a simple hierarchical fashion. Cyclic references and multiple paths to the same data can cause endless issues for common scripted methods of traversing hierarchical layouts. File system and tool developers must carefully consider the implications of their selected topology on the ways in which it is likely to be used.

An important consideration is the tradeoff between overloading file system semantics to achieve required functionality, and the user expectations for the file system operation that was overloaded. Our experience is that it is often a mistake to violate the rule of least surprise. However, there are situations in which correctness demands a different behavior than what is typically expected when interacting with files. An example is the "clone" file which is used by certain Plan 9 synthetic interfaces to allocate resources. To avoid race conditions in the automatic garbage collection of the resources which clone allocates, the file descriptor returned by opening a clone file actually points to the control file interface within a subdirectory representing the resource instance which opening the clone file allocated. The alternative is a multi-stage provision, access, and release sequence which opens up the potential for race conditions and complicates error handling and recovery.

A commonly perceived challenge is the performance overhead of the additional layer(s) of indirection as well as parsing string-based control commands. In practice much of this overhead is either insignificant or avoidable. However, the implementors of the synthetic file server infrastructures as well as the services themselves must be aware of performance overheads in order to avoid unnecessary copying of data, expensive parsing and irrelevant synchronization. Of course these performance pitfalls are common to many such service provider applications.

Like the entries in traditional file systems which serve as interfaces to disk files, extant synthetic file systems only permit a small set of file properties, largely structured around ideas that still date back to disk file systems (e.g., size, access rights and modification times of entries in the file name space). One direction of further evolution is the consideration of synthetic file systems in which the properties of individual entries in the file name space may have an arbitrary *structured type*, built from a small set of primitive types. This idea has been explored to a limited extent as the basic underpinning of the runtime system for a programming language targeted at heterogeneous concurrent hardware platforms [17]. The idea of generalized structured types for the *names* of entries in a file name space is related to the idea of types for the *data* in a file store and object-based stor-

age [13]; the difference lies in the fact that the name of an entry can foreseeably have a different type from the items obtained by interaction with such a name.

5 Conclusion

Service Oriented Architectures and RESTful interfaces are becoming a dominant paradigm in applications development and integration. We have compared this approach with that of synthetic file systems and proposed a systems software approach which we facilitates more natural interaction and composition of these emerging applications, tools, and services.

We are currently exploring the use of synthetic file systems for a wide range of systems including cloud provisioning and management, large scale simulation infrastructures [18], semi-supervised machine learning for translation and semantic search(in progress), and as the distributed systems infrastructure for high-performance computing [20]. As a foundation for these projects we are developing tools to help with the design and implementation of synthetic file servers. We are also looking into mechanisms which mitigate the performance overhead of going through an operating systems file interface.

This material is based upon work supported by the Department of Energy under Award Number DE-FG02-08ER25851.

References

[1] F.J. Ballesteros, P. de las Heras, E. Soriano, G. Guardiola, and S. Lalis. The Octopus: Towards Building Distributed Smart Spaces by Centralizing Everything.

[2] A. Barak and O. La'adan. The MOSIX multicomputer operating system for high performance cluster computing. *Future Generation Computer Systems*, 13(4-5):361–372, 1998.

[3] A. Bergmann. Linux on Cell Broadband Engine status update. In *Proceedings of the Linux Symposium*, pages 21–28, 2007.

[4] M. Burrows et al. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th OSDI Conference*, 2006.

[5] N. Evans. Representing disparate resources by layering namespaces. *the Second International Workshop for Plan Proceedings*, 2007.

[6] R. Fielding. *Representational State Transfer (REST)*. PhD thesis, PhD thesis, available at: http://www. ics. uci. edu/fielding/pubs/dissertation/rest_arch_style. htm, 2000.

[7] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2007 conference on EuroSys*, pages 59–72. ACM Press New York, NY, USA, 2007.

[8] TJ Killian. Processes as Files. In *USENIX Summer Conference Proceedings*, 1984.

[9] B. Martin. Everything is a virtual filesystem: libferris. In *Proceedings of the 2007 Ottawa Linux Symposium (OLS 2007)*, volume 2, pages 223–227, Ottawa, Canada, June 2007.

[10] M.D. McIlroy. Pipes and filters. *Internal Bell Labs memo, original title lost*, 11, 1964.

[11] R. Minnich, A. Mirtchovski, and L. Ionkov. XCPU: a new, 9p-based, process management system for clusters and grids. *Cluster 2006*, 2006.

[12] N. Mitra et al. SOAP Version 1.2 Part 0: Primer. *W3C Recommendation*, 24, 2003.

[13] D. Nagle, M. E. Factor, S. Iren, D. Naor, E. Riedel, O. Rodeh, and J. Satran. The ansi t10 object-based storage standard and current implementations. *IBM J. Res. Dev.*, 52(4):401–411, 2008.

[14] R. Pike, D. Presotto, S. Dorward, B. Flandrena, K. Thompson, H. Trickey, and P. Winterbottom. Plan 9 from Bell Labs. *COMPUTING SYSTEMS*, 8:221–221, 1995.

[15] R. Pike, D. Presotto, S. Dorward, DM Ritchie, H. Trickey, and P. Winterbottom. The Inferno operating system. *Bell Labs Technical Journal*, 2, 1997.

[16] R. Pike, D. Presotto, K. Thompson, H. Trickey, and P. Winterbottom. The use of name spaces in plan 9. In *Proceedings of the 5th workshop on ACM SIGOPS European workshop: Models and paradigms for distributed systems structuring*, pages 1–5. ACM Press New York, NY, USA, 1992.

[17] P. Stanley-Marbell and D. Marculescu. A Programming Model and Language Implementation for Concurrent Failure-Prone Hardware. In *Proceedings of the 2nd Workshop on Programming Models for Ubiquitous Parallelism, PMUP '06*, September 2006.

[18] Phillip Stanley-Marbell. Implementation of a distributed full-system simulation framework as a filesystem server. In *Proceedings of the First International Workshop on Plan 9*, 2006.

[19] E. Thomas. Service-Oriented Architecture: Concepts, Technology, and Design, 2005.

[20] E. Van Hensbergen, C. Forsyth, J. McKie, and R. Minnich. Holistic aggregate resource environment. 2008.