5-1-2013

A Universal Framework for (nearly) Arbitrary Dynamic Languages

Shad Sterling
Georgia State University

A UNIVERSAL FRAMEWORK FOR (NEARLY) ARBITRARY DYNAMIC LANGUAGES

(A THEORETICAL STEP TOWARD UNIFYING DYNAMIC LANGUAGE FRAMEWORKS AND OPERATING SYSTEMS)

by

SHAD STERLING

Under the Direction of Rajshekhar Sunderraman

ABSTRACT

Today's dynamic language systems have grown to include features that resemble features of operating systems. It may be possible to improve on both by unifying a language system with an operating system. Complete unification does not appear possible in the near-term, so an intermediate system is described. This intermediate system uses a common call graph to allow components in arbitrary languages to interact as easily as components in the same language. Potential benefits of such a system include significant improvements in interoperability, improved reusability and backward compatibility, simplification of debugging and some administrative tasks, and distribution over a cluster without any changes to application code.

INDEX WORDS: Dynamic Programming Languages Framework Operating Systems Distributed Interoperability

A UNIVERSAL FRAMEWORK FOR (NEARLY) ARBITRARY DYNAMIC LANGUAGES

(A THEORETICAL STEP TOWARD UNIFYING DYNAMIC LANGUAGE FRAMEWORKS AND OPERATING SYSTEMS)

by

Shad Sterling

An Honors Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of

Bachelor of Science

in the College of Arts and Sciences

Georgia State University

2013

TABLE OF CONTENTS

1. INTRODUCTION

1a. Programming Languages

Programming languages have changed significantly over the last few decades. In the 1970s the C programming language was significant milestone. It was not tied to a specific hardware platform, but provided access all the basic operations of almost any CPU. It had language features that allow a programmer to organize their code and the data it handles, but that do not impose any run-time overhead when used.

The C programming language both allows and requires programmers to specify every basic operation the program is to perform. This costs a significant amount of programmer time, but allows optimization of computer time that often rivals programming in assembly language. When C was new, computation was much more expensive than programmer time, which made requirement of specifying every basic operation an advantage.

Since then the cost of computation has decreased, and in most contexts the requirement that programmer time be spent specifying each operation is no longer advantageous. Many newer languages have language features that determine the basic operations at run-time, increasing the run-time computation required and decreasing the programmer time required.

One such feature that has become nearly ubiquitous in mainstream languages is polymorphism. In some cases static analysis can allow polymorphic statements to be optimized into one morphology, or some specific small number of possible morphologies, but it is now a common language feature to allow programmers to specify higher-level operations by name and have the expansion of those names into basic operations determined at run-time.

The determination of morphology is made based on the data to which the named operation is applied. The common organizational structure used to make this determination is to organize data into objects, and have the mapping of names to basic operations determined by the object to which the name is applied. In most contexts this can be referred to as "invoking a method on an object" (with operators understood to be methods), but some newer languages prefer to call it sending signals or passing messages[1]. I will refer to these mechanisms as inter-object communication.

The execution of programs in these languages is also different from older languages. No longer compiled to machine code before execution begins, nor interpreted straight from the source, the common mechanism today is to compile at launch-time to an intermediate code, which is then interpreted by a language-specific "virtual machine," which may further "just-in-time" compile individual methods at run-time into local machine code. All of the components involved in executing a program in one of these languages form a "language system". Some language systems can re-use the intermediate code and machine code, rather than repeating the work every time a program is run, without any special effort on the part of the programmer or user.

Typically, a language system runs one process for each instance of each program in the corresponding language. Within each instance, the language system manages a set of objects, conceptually distinct entities, which are protected by the mechanisms of the language system, and which can be created and destroyed. The language system provides a mechanism for inter-object communication, and for interacting with external entities such as other processes, peripheral devices, users, and networks. Execution of a program begins with invocation of some

initial method, which can create other objects, communicate with them, and interact with external entities.

1b. Operating Systems

While operating systems have also changed significantly over the past few decades, the relevant parts have changed fairly little. All general-purpose operating systems still support the functionality of the C standard library, which was developed along with the C language and early versions of UNIX. Modern operating systems have changed how API functions are implemented, and added many features and services, but have retained some fundamental properties in common with early preemptive multitasking systems.

Typically, a separate instance of the operating system runs on each mainboard. Within each instance, the OS manages a set of processes, conceptually distinct entities, which are protected by the mechanisms of the OS and CPU, and which can be created and destroyed. The OS provides mechanisms for inter-process communication, and for interacting with external entities such as peripheral devices, users, and networks. Use of the OS begins with the launch of some initial process, which can create other processes, communicate with them, and interact with external entities.

Viewed this way there are some striking similarities between the organization of an operating system and of a language system – both manage and a dynamic set of distinct entities, provide mechanisms for communication between them, and provide access to external entities. Building on the similarities, it may be possible to create a unified system that is at once a language system and an operating system. Such a system might not be a dramatic departure from existing systems, but might have some advantages over the separate systems we have today.

1c. Grand Unified Computing

Overhead

Operating systems imposes overhead primarily when making a request of the OS itself; interpreters impose overhead on every operation; JITs impose overhead only when using inter-object communication. An inlining JIT could significantly reduce inter-object communication.

When a program in one of these modern languages consists of a single pre-compiled method (or has been inlined into a single method), and this method never uses inter-object communication or any other language system facility, the modern language imposes no run-time overhead – some at launch-time, end-time, and exception-time, but none during normal operation. Such a program is very nearly equivalent to a traditional program running on the host OS.

An optimal language system would impose run-time overhead only when a program uses language system facilities in a way that cannot be statically optimized out. A language system which allows a programmer to configure and confirm optimizations would allow any particular program to be made as much like a traditional program as it needs to be.

On a unified system, language system facilities are OS facilities, so the launch-time, end-time, and exception-time overhead is merged. The overhead imposed by a unified system could be all but indistinguishable from the overhead imposed by current OSs.

External Entities

Most language systems map access to external entities through the language library API to the host OS API. On a unified system the underlying OS mechanisms would not need to change, and there would be no need for mapping, so this access would be essentially the same but with slightly less overhead than in current language systems.

Generality

Language Systems are less general than Operating Systems. In the simple cases, they only run programs in one language. In less simple cases, there are multiple languages that are compiled to a common intermediate code (e.g. JVM and .NET bytecodes[2]). But even those "multi-language systems" are less general than an OS, which must run programs precompiled to machine code from any language whatsoever. To be as general as an OS, a multi-language system would have to be able to accommodate any language whatsoever. A unified system could not be an ordinary language system or multi-language system, it would have to be a universal language system.

Interoperability

Within any language system, a programmer can take advantage of any existing code already written in that language, but in most cases using code written in any other language is problematic. In an OS, any program can make use of any service or shared library available for that OS, regardless of source language, but a lack of common conventions for interaction protocols limits cross-language use in practice. A unified system could establish conventions that would make cross-language library access more practical.

A perfect solution would expose any shared library to any language as a set of apparently native functions – as methods on an object representing the library. This would be a natural feature of a universal language system, regardless of OS unification.

Stability

Operating systems are expected to run for up to several years between reboots[3,4], and are generally good at keeping the OS useful when individual processes fail. Language system

processes are not expected to remain running even for months at a time, and generally do not have good mechanisms to keep some objects available after others have failed. Buggy object-oriented programs are likely to have "object leaks," which most language systems do not have any mechanisms to constrain. Any OS can handle a language system process with a memory leak, but OS features to handle a "process leak" (aka fork bomb) are lacking. A unified system would have to be a universal language system with absolutely perfect garbage collection which is immune to both object leaks and process leaks. This does not appear to be practical in the near term.

Security

In the most general form of unified system, everything in the OS would appear to be in the same language system process; there would be one set of objects, which would appear to be within all programs at all times. Providing security in this environment would resemble a conventional language system providing security between different parts of the same program. Current language systems do not have any mechanisms for that sort of security.

Some of existing OS security mechanisms would work in a unified system, but some new mechanisms would need to be developed. Security in general is hard, and any new mechanisms need to be designed carefully, tested thoroughly, and expected to fail anyway when first exposed to the wild. Getting security right on a unified system would be a long term project, so is not practical in the near term.

1d. Language Loader

Creating a unified system might be possible and worth pursuing, but does not appear to be practical in the near term. At the least, establishing security and long-term stability will not

be done soon. In the short term, creating a universal language system that runs on existing OSs is more likely to be viable.

The following sections give an overview of one possible universal language system. The next section starts with the key aspects of how the system works: the coordinating backplane, object model, method execution mechanism, data exchange model, and the call graph. Following that is a section with additional detail on selected topics, and then a section of more speculative goals such a system might achieve in the distant future.

2. KEY ASPECTS

2a. The Backplane

Almost everything in the system is an object. The things that cannot be objects – such as the mechanism to invoke methods on other objects – are part of the software backplane. The backplane includes implementations of some special objects and methods which are addressed in the usual way but do not impose the overhead of the usual object creation or method invocation mechanisms. Interfaces to host the OS, CPU features, and peripheral busses are part of the backplane.

The primary function of the backplane is to facilitate interaction between objects, including objects that represent external entities. All such interactions are represented in the call graph, a set of special objects that represents all computations in progress and the dependencies between them. The backplane handles system requests made within methods, and edits the call graph appropriately, such as expanding the call graph when invoking a method and contracting the call graph when a method returns.

Most external entities are not handled directly by the backplane, but by ordinary objects which use the special peripheral bus objects to communicate with a particular external entity. The backplane does handle hardware interrupts and host OS signals, by some combination of event triggering (which invokes any methods associated with the triggered event) and exception handling (which invokes an exception handlers associated with the method invocation that triggered the exception).

2b. Objects

Data in the system is organized into objects, which consist primarily of some content, some instance variables, attached methods, and attributes. The content of an object is a block of arbitrary data, which is accessible by methods invoked on that object. The variables are a mapping of identifying symbols to arbitrary values, accessible by methods invoked on that object. The attached methods are a mapping of identifying symbols to method objects, which can be invoked on that object by ordinary calls. The attributes are a mapping of identifying symbols to arbitrary values which are readable by methods on other objects without invoking another method. A mapping from identifying symbols to any values effectively assigns names to those values, so this arrangement provides names for variables, methods, and attributes, none of which have intrinsic names.

While the content of an object appears to a method as a single contiguous address space, the actual storage of the object may not match. It may be sparse or fragmented (as a file may be on disk, or a process' address space in physical memory), and it may contain ranges which map to other objects. Mapping one object into another can be used for containment (such as embedding an image in a document) and for sharing (such as memory-mapped files or shared library functions). System requests that create objects of existing data will generally leave the data in place and create a mapped object.

In addition to the attached methods, each object may have both ancestors and heritable methods. The ancestors are an ordered list of objects whose heritable methods will be invoked when the identifying symbol does not match any attached method. The heritable methods are a mapping of identifying symbols to method objects that can be invoked on descendents. In common terms, every class is an object, and every object is a class. Method invocation searches

by symbol through the object-specific methods, then traverses the ancestor list in order searching the heritable methods. A particular method can be both attached and heritable, and an object can be it's own ancestor.

This model of attributes and inheritance will not support every possible pattern, but does support most features of common models. Having attributes in addition to accessor methods is important for minimizing overhead, but is less flexible than the options available in other languages[5]. Accessor methods can provide the same flexibility, but adds the overhead of a method invocation (unless an inlining JIT removes it). The linear inheritance avoids the inheritance diamond problem by forcing a total ordering of ancestors.

2c. Method Execution

Methods are objects which have as their content is some sort of executable code, and attributes specifying what is needed for the method to function. Any invocation of any method is represented in the call graph by an invocation object, which encapsulates the local memory and execution state of the invocation, as well as some context information. The object on which a method is invoked is the "target object" of the invocation. For methods which can be run directly on hardware the invocation object is analogous to OS process information, and the backplane uses internal (or host OS) mechanisms to handle execution. For methods that are not in local machine code, the backplane must invoke an execution handler from an appropriate language module.

An execution handler is an interpreter method that acts as a virtual CPU to execute code that cannot be executed directly by any available CPU. Invocation of the interpreter method is represented in the call graph by an additional invocation object, which encapsulates the state of

the interpreter. Invocations of execution handlers have access to modify the state of the invocation they are handling, and to make system requests on behalf of the invocation they are handling. Since execution handlers are methods, any execution handler that cannot be executed by an available CPU can be executed by another execution handler. In addition to executing a method, a language module can also compile or translate a method to generate new variants which the backplane may choose to use for future invocations.

The execution context associated with an invocation controls execution priority and access rights, and provides references to API objects, filesystems, external entities, and neighbors in the call graph. By default, a new invocation inherits the execution context from which it was invoked, but a new context can be specified at invocation. The execution context resembles a generalization of environment variables, no longer restricted to strings, which can be used to provide or restrict access to much of the system. Modified execution contexts can be used to "sandbox" untrusted methods, create segregated environments for administrative purposes (like "chroot virtual machines"[6]), expose APIs to plugin methods, provide deprecated APIs for backward compatibility, and so on. The method only "knows" what's in its execution context, it does not know how isolated it is from any other contexts that might be on the same system.

2d. Data Exchange

While there is no restriction on how methods handle data internally, or how they modify the content of the target object, data passed to system requests or stored in collection objects with special implementations will carry type information. Available types will include common machine-supported primitive types, IEEE standard types, an object reference type, and an unknown type (to be used for data of any unsupported type). Including type information allows

run-time type checking on data exchanged through the backplane, and allows reference tracking for storage management and garbage collection.

System requests that invoke a method take a single value as a parameter, and returning from a method also takes a single value. Passing multiple parameters requires passing a single value which is an object reference to a collection object containing the parameters. When the collection object has a special implementation, creating the collection and accessing members of the collection has less overhead than the usual object creation or method invocation mechanisms.

The backplane will include special implementations of several kinds of collections, including arrays, mappings, and ordered mappings, as well as other kinds of critical objects including synchronizers, strings, identifying symbols, and those used in the call graph. Collections with special implementations will be used for instance variables, attached methods, attributes, ancestors, and all other metadata necessary for the system to function. These must not only have special implementations, but must restrict the values that may be included – for example, attached methods are a map from identifying symbol objects to method objects. Similar restrictions may be applied to any collection using the special implementation.

2e. Call Graph

The call graph is an acyclic simple directed graph with invocation objects as nodes. Most edges in the call graph are "result" edges, which represent dependance on return value. A result edge from invocation A to invocation B means A depends on the return value of B; A is said to be a "dependent" of B, and B a "dependency" of A, and the edge direction is said to be "downward." Dependance does not imply blocking; a dependent will block on a result edge if and only if it requests the return value before the dependency returns. The subgraph including

only result edges is the partially ordered computation-dependance graph; the downward subgraph from any node contains every computation on which that node depends, and the upward subgraph contains every computation which depends on that node.

Other edges in the graph include “emulator” edges and “queue” edges. Emulator edges connect an emulated invocation to the invocation of its execution handler. In the subgraph including only emulator edges, each node has indegree and outdegree at most 1; the emulator subgraph consists of disjoint chains. Queue edges represent synchronization blocks. The dependent on a queue edge blocks until the dependency returns, regardless of whether there is any result path between them.

This graph only represents calls made through the backplane; what the backplane sees as a single method may be arbitrarily complex. Most existing applications could be run as single methods, using the backplane only where existing OS API calls are used, with very little change in overhead imposed by API calls. Every object in the call graph has a special implementation, so the overhead on changes to the call graph such as invocation and return is similar to that of system requests on other operating systems or language systems.

Tail calls, continuation passing style, and other less common patterns can be implemented by making appropriate edits to the call graph. Inspection tools can be useful for debugging, administration, and users interested in learning how their software works. When the necessary execution handlers are available, downward subgraphs can be pruned from one system and grafted onto another system with a different hardware architecture and/or host OS. Bug reports could include a copy of the relevant downward subgraph, which could be inspected and/or

resumed by developers. Hibernation would amount to pausing the entire graph, and quick startup would be resuming a saved initial graph.

3. SELECTED DETAILS

3a. Object Classification

Restrictions on objects included in special collections can be based on classification of objects. Since objects are not defined as being a member of one specific class, classification of objects must be done differently, and can be done in more than one way. The closest to checking class membership is checking ancestry; testing whether a particular object (the "class" object) is in the ancestor list closely resembles testing for class or subclass membership. Classification can also be done by "quack check," testing than an object will respond to a set of method names, which is slightly stronger than "duck typing" with no checking at all. Method objects can be recognized, and classified by what is needed for the method to function. Objects can also be classified by which (if any) special implementation is used.

3b. Objects with Special Implementations

Special implementations are given to objects that are necessary for the system to function, including objects representing data describing other objects, objects related to hardware or host OS, management information such as the call graph and synchronizers, and objects used pass data to system requests. Some of these implementations can be used for other purposes, and individual objects with these implementations can be created using system requests and modified as needed (similar to deriving from an intrinsic class in some dynamic languages). Implementations will include several collection types, strings, identifying symbols, and blobs of arbitrary data.

The terminology for collections varies, but implementations will be available for unordered sets, ordered sets (arrays), indexed sets (associative arrays), and ordered indexed sets.

any of which may carry restrictions, default values (for otherwise invalid indices or keys), and/or a fallback collection to which any failing lookup is forwarded. Collections that use optional features may use different special implementations, so the availability of unused features does not increase overhead. Strings are sequences of unicode graphemes, in which each entry is a base character and any combined (accent) marks. String objects behave as though there is no encoding; an encoding must be specified for every import or export. Identifying symbols resemble immutable strings, but guarantee at most one symbol object per unique sequence; symbols can be compared by comparing object IDs. (These identifying symbols resemble Symbol objects in Ruby[7], or atoms returned by the GlobalAddAtom function in the Win32 API[8].)

3c. Synchronization

The basic synchronization mechanism is the combination of queue edges and blocking objects. Only one invocation can be active per blocking object. When a method on a blocking object invoked another method on the same blocking object, it blocks until the new invocation returns. When an invocation on a blocking object is active, any new invocations on the same object (not made by the active invocation) are blocked by a new queue edge from the new invocation to the active invocation. General-purpose semaphores and mutexes with special implementations are provided as blocking objects, but a blocking object that represents a specific concurrency controlled resource could implement an arbitrary control scheme. Queue edges in the call graph created by invoking methods on blocking objects form disjoint chains, one per blocking object. In general, the graph theory approach to deadlock prevention can take

advantage of the queue edge subgraph. A blocking object with no named methods and a default method would amount to the message queue paradigm used in some APIs.

3d. Files & Storage

Files are objects that conform to a specified interface; the content of the object is the content of the file, and the file metadata are attributes of the object. Since the root of a file tree is an object, multiple disjoint file trees may exist in the same system, with access controlled by execution contexts. Files stored on a disk are represented as file objects, but file objects do not necessarily represent files on disks. The recent trend of treating "apps" as locations could easily be followed, the apps would only need to generate objects bearing the specified interface. A filesystem would be a module that implements a particular method of storing a file tree (or file graph) as mapped objects within another object, often an object representing a data storage hardware device, often an object representing an archive file. Objects that are not files may also be stored as files in a filesystem, but a more general object storage system would be preferred, and may follow the SCSI OSD standard[9] and/or resemble a ZFS object store[10].

3e. Hardware Devices

When running on a host OS, the backplane must include implementations of objects representing hardware devices accessible through the host OS. When running as an OS, the backplane must include implementations of objects representing core hardware, at least whatever busses are used to connect directly to the CPUs. In either case, any bus device represented by an object with a special implementation acts as a root of a device object tree. The device object tree essentially mirrors the physical device tree, with the root representing a bus accessible via the host OS or a CPU bus, connected busses adding branches, and terminal devices as leaves.

Objects representing terminal devices communicate through the object representing the associated bus device, the signals ultimately passing through the root object to the host OS or CPU bus. Objects representing disconnected devices do not just disappear, they go into a disconnected state, and devices which can be uniquely identified will become reconnected when the same device is reconnected, even if the path through the bus tree has changed.

3f. System Requests

How system requests are invoked and implemented depends on the platform; what system requests do does not. In a minimal case, system requests must be used to access collections to utilize multiple parameters, create collections, strings, and/or symbols to pass as parameters, invoke methods on other objects, and to return a value. The base cases of creating a new object are to map existing data into the new object, or to create a new object which is empty. Existing data can be in the body, the local memory, or the target object of the invocation making the request. Invocations are by default asynchronous; the invoke system request returns a reference to the new invocation object, the return value of which can be retrieved by the potentially blocking await system request. Additional requests support tail calls and continuation style. System requests do not raise exceptions, they return error values.

3g. Scheduling

CPU time is allocated to active invocation objects. Active invocations form the lower region of the call graph, which could (in principle) be the entire graph. When possible, a low-priority invocation which is blocking a high-priority invocation will be promoted. More generally, resource usage can be attributed to higher nodes in the graph, which can allow scheduling choices to isolate contention in some subtrees and ensure responsiveness in others.

4. DISTANT GOALS

4a. Run-Time Optimizations

Language modules provide code analysis methods as well as execution handler methods. The code analysis is language-specific, but provides information that can be used to apply optimizations to any language at run-time. For example, any referentially-transparent method can be automatically memoized when it reaches some popularity threshold, regardless of the language used. Language modules may also include JIT compilers; the work to create JIT compilers that can inline methods from other languages would happen here.

4b. Backward Compatibility

Language modules provide compatibility for code in a particular language. System request translation modules provide compatibility for code using a different set of system requests. These can be used to execute code intended for a different operating system, or for a different version of the backplane. As with nesting of execution methods, translation modules can be nested, potentially providing deep backward compatibility – if new modules are created as old systems are obsoleted, code could remain usable long after the language used and hardware expected have gone out of use.

4c. Transparent Clustering

In general, the backplane does not guarantee that any methods invoked are executed locally, or that objects referenced are stored locally. Any computation involving multiple invocations may be distributed arbitrarily over a cluster. The common case of a multicore CPU on a multisocket mainboard with multiple GPUs connected to expansion busses is handled as a simple low-latency cluster. JIT compiling to GPU machine code will allow GPU power to be

applied to any computation (possibly trading lower power-efficiency for lower total execution time). Wider clusters will require additional setup, after which any invocation may be executed anywhere in the cluster.

4d. Data Integrity

An additional option for object classification is by content type. Objects marked with a content type are protected from modification by methods not also marked as compatible with that content type. (Other methods may be invoked, but be unable to modify the object.) Recognized content types will include all specially implemented types and the IANA media types (fka MIME types)[11]. Eventually, object modifications will be organized into transactions.

5. CONCLUSIONS

If the system works as intended, it will have several notable features which are absent or more troublesome in most existing systems. None of these are unique, but their generality, the combination of features, and the way features are enabled may make this a more powerful system. Some notable features include:

- The call graph can be inspected for reflection, debugging, administration, or user exploration/education
- Nested emulation with execution handler methods and system request translators allows broad compatibility and deep backward compatibility
- An application's portion of the call graph can be paused and resumed, or pruned from one machine and grafted on another with a different hardware architecture and/or host OS.
- Resource utilization tracking by dependent invocations simplifies usefully identifying resource hogs.
- Embedding allows space efficiency, and can help with document maintenance when editing multiply-embedded data in one place affects all embeds of the same object.
- Files can be located in apps, archive files, filesystems, or a general object store

REFERENCES

1. Smalltalk, in 1980, may have been the first object-oriented language to use the terminology of message passing; messages are described as more central than objects; Alan Kay. “Prototypes vs Classes.” *Squeak developers list*, The Squeak Foundation, 1998; http://lists.squeakfoundation.org/pipermail/squeak-dev/1998-October/017019.html retrieved July 2013
2. Authors do not appear to maintain lists of their competitors; several languages that target both JVM bytecode and Microsoft CIL are listed by Wikipedia: http://en.wikipedia.org/wiki/List_of_JVM_languages retrieved July 2013, http://en.wikipedia.org/wiki/List_of_CLI_languages retrieved July 2013
3. Novell collects reports of high uptimes from their customers, all uptimes listed here are over one year; Dave Kearns. “Marathon Servers.” Network World, 2005; http://www.networkworld.com/newsletters/netware/2005/1128nw2.html retrieved July 2013
4. The record uptime for a server may the 16 years reported here; Peter Bright. “Epic uptime achievement unlocked. Can you beat 16 years?” Ars Technica, 2013; http://arstechnica.com/information-technology/2013/03/epic-uptime-achievement-can-you-beat-16-years/ retrieved July 2013

5. A summary of access modifiers in C++, Java, C#, and Smalltalk;

   Martin Fowler. "AccessModifier." (personal blog), 2003;

   http://martinfowler.com/bliki/AccessModifier.html retrieved July 2013

6. A howto for using chroot to create an alternate development environment on Ubuntu;

   "Using Chroots." *Ubuntu App Developer*, Canonical Ltd.:

   http://developer.ubuntu.com/packaging/html/chroots.html retrieved July 2013

7. Documentation of the Ruby Symbol class;

   "Symbol." *Help and documentation for the Ruby programming language*, The Ruby Documentation Project;

   http://www.ruby-doc.org/core-2.0/Symbol.html retrieved July 2013

8. Documentation of the Win32 GlobalAddAtom function;

   "GlobalAddAtom function." *Windows Dev Center*, Microsoft, 2013;

   http://msdn.microsoft.com/en-us/library/windows/desktop/ms649060(v=vs.85).aspx

   retrieved July 2013

9. Christian Bandulet. "Object-Based Storage Devices." Oracle, 2007;

   http://www.oracle.com/technetwork/server-storage/solaris/osd-142183.html

   retrieved July 2013

10. O. Rodeh, A. Teperman. "zFS – A Scalable Distributed File System Using Object Disks." *Conference on Mass Storage Systems and Technologies*, IEEE/NASA, 2003;

    http://storageconference.net/2003/papers/29-Rodeh-zFS.pdf retrieved July 2013

11. The Internet Assigned Numbers Authority (IANA) maintains a list of "Media Types" (formerly known as "MIME types"), which are standard names for static data encoding

formats;

“MIME Media Types.” Internet Assigned Numbers Authority, 2012;

http://www.iana.org/assignments/media-types retrieved May 2012