Introduction to Resource-Oriented Computing

Part I

1060® Research white paper series:

Introduction to Resource-Oriented Computing – Part I

Version: 1.0.2

July 12, 2007

Executive Summary

This white paper introduces Resource-Oriented Computing (ROC) and explores the reasons why it provides an economically compelling and technically elegant computing platform. The paper presents the axioms of ROC, discusses historical examples and examines a complete resource-oriented computing platform.

This paper makes statements about performance and the economics of system engineering that will sound to many like well worn marketing hype. To demonstrate that these statements are in fact simple facts, the paper introduces and builds upon a foundation of fundamental principles. It is likely that these principles will challenge your understanding of the nature of computation.

Resource-oriented computing is a new computing model with an old history. ROC is concerned first and foremost with information processing. It emphasizes logical information sources, uses, and transformation, ahead of physical code, data, and programming languages. The change of primary focus from languages to information represents a departure from an evolutionary path that has led from machine code to assembler, procedural, modular, and then object-oriented programming. Whilst this evolution has yielded significant improvements in productivity and expressiveness, ties to the physical computing layer (such as physical addresses) have restrained a leap forward to a truly simple logically-based computational model.

Resource-oriented computing offers this new simplified computing model. While it may seem that a model separated from direct association with the physical computing layer would be inefficient, hard evidence indicates the opposite. Resource-oriented systems typically run three to four times faster than equivalent systems written in Java J2EE and Microsoft .Net. Resource-oriented systems scale with CPUs for the same reasons that web sites scale with load balancing.

The economics of resource-oriented computing are compelling. Systems require ten to one hundred times less code and application development time is measured in weeks instead of months and years. Life-cycle costs are dramatically lower because the resulting systems manage complexity and are inherently more flexible.

Resource-oriented computing is based on simple principles and is easy to learn. This white paper provides a high-level introduction to the major concepts. More information is available from 1060 Research at www.1060research.com

Introduction

Resource-oriented computing (ROC) is a simple fundamental model for describing, designing, and implementing software and software systems. Its simplicity derives from a small set of principles discovered through research initially started at Hewlett-Packard Laboratories and continued and expanded by 1060 Research. Resource-oriented computing is simple, *not* simplistic. Powerful production systems have been built in a fraction of the time required by an equivalent object-oriented solution. It is consistently shown that resource-oriented systems require one to two orders less code, typically perform three to four times faster, are inherently more flexible and robust to change, and require zero modifications to scale on platforms as the number of CPU cores is increased. But, to be able to differentiate these statements as cold facts and not industry hype, we must first explore our fundamental understanding and preconceptions of what computation is.

If you prefer to study some examples of real resource oriented systems please skip ahead to the "Examples of Resource Oriented Systems" section. If you want to think about the fundamental basis of computation stick with it - if the going gets tough you might want to step off to look at the examples too!

Resource Oriented Computing Fundamentals

A *resource* is a set of information. Specifically, in resource-oriented computing, resources are treated as abstracts; that is, a resource is a *Platonic*[1] concept of the information that is the subject of a computation process. At the physical level, a ROC system processes resource-representations, executes transformations and, in so doing, computes new resources. In this respect ROC is no different to any other computational model – computation is performed to collate and reveal new information.

The shift in thinking introduced by ROC allows one to directly consider the abstract world of resources and in so doing, step-up and away from the physical implementation details of languages, object models and code. With ROC one designs and develops information processing systems by working on a new plane that is fundamentally logical and abstract. On first introduction such talk can sound like a bizarre and philosophical basis for a software development paradigm! *Our purpose in this white paper is to demonstrate that by placing computation on a solid and fundamental resource-centered foundation we can obtain huge practical returns and reveal a world of software development that lives in close harmony with the set-theoretic fabric of computation theory and that is unencumbered by the intricacies and escalating complexity of physical code.*

Having embarked on an apparently crazy philosophical mission, let us consider our intuitive understanding of abstract resources and see how this is something we are actually very familiar with outside of the software realm. For example, the story "Jack and the Beanstalk" is an abstraction that can have many physical realizations. It could be represented as a printed hard cover book, a printed paperback book, an audio book in MP3, a word-of-mouth story; and for each, there can be an English language version, a French or Spanish language version, etc. When we refer to "Jack and the Beanstalk" we usually mean the abstract idea of the story, which covers the potentially infinite set of possible representations. In fact within the software realm we encounter this all the time, for example, the abstraction "People Registered for the Conference" refers to a particular set of people. In a computer system this can be represented as a database table, an XML document, a CSV file, or as a collection of linked objects; and for each, there can be Java language version, Ruby, Python, Lisp, etc. In all cases, the information resource is the same and the computer user viewing their display or reading the printed attendee list is totally indifferent to the choice of physical representation; their only concerns are about correctness and availability of the information.

1 Harte, "Plato on Parts and Wholes", Oxford University Press ISBN: 0198236751

To get a little more formal, we can define the first axiom of resource-oriented computing as:

A resource is an abstract set of information

In order to be able to compute anything we must be able to identify what it is we are computing. In programming languages we understand that we must assign transient logical labels to information resources held in memory – we call these variables. In resource-oriented computing we have stepped off the physical plane of languages and their close ties to physical memory and onto a logical plane. We can therefore choose to identify a resource using any suitable logical identifier. The second axiom of resource-oriented computing is therefore:

Each resource may be identified by one or more logical identifiers

This has a feeling of being very abstract! What do we mean? Let's reconsider the real-world example, the children's story identified by the title "Jack and the Beanstalk". A particular representation of the story, a book printed by a publisher, could also be considered as a manifestation of the resource and it would be identified by an ISBN. In this example, each physical printed copy is a representation, while the ISBN identifies the abstract idea of the printed book. There is a potentially-infinite set of identifiers for the resource that may be useful – for example, different ISBNs for hardback and paperback etc. A key understanding is that an identifier has value when it can be used within a specific *context* to *resolve* a physical representation of the information resource. For example, Amazon.com or the public library provide an information context in which an ISBN can be used to resolve a physical copy of the story. This suggests the third axiom of ROC:

A logical identifier may be resolved within an information-context to obtain a physical resource-representation

Resource-oriented computing is concerned with constructing software systems which usefully resolve logical identifiers to obtain physical representations. In resource-oriented computing we can define a *context* as an information-space that contains a set of identifiable resources. It can be convenient to use the more intuitive term *address-space* and to think of *resource-identifiers* as addresses in the space – but be careful not to confuse these with the historic usage of the terms for physical-addressing within computing systems such as a pointer to a memory location; resource-oriented computing is concerned with *logical* identifiers and *logical* addressing. Using our book example, a useful address space is the set of books that can be identified by an ISBN. Within the software realm, a resource-oriented system can use any suitable addressing system and, as we

will see later, a particularly useful addressing system is the Uniform Resource Identifier (URI). From this discussion we can introduce the fourth axiom of ROC:

Computation is the reification of a resource to a physical resource-representation

When a resource is requested within a resource-oriented system a resource-identifier is resolved within a context and a concrete, immutable representation is provided. It is important to highlight the immutability requirement of representations. We have said that there may be many resource-identifiers for the same logical resource – therefore it is essential in a heterogeneous logical computing system that physical representations are immune to side-effects – that is, a resource's information-set cannot be changed by modifying a representation. Fundamental changes to a resource are achieved only by acting on the primary source of the resource itself. To use the book metaphor, if we were to take a copy of Jack and the Beanstalk and rewrite the information related to the concept of the 'Bean' and the 'Beanstalk' in order to tell a story about 'Rocket Fuel' and a 'Magic Rocket' then the modified representation would not correspond with the 'Jack and the Beanstalk' identifier and the reader would be confused. The fifth axiom is:

Resource representations are immutable

A representation is current as of the moment it is requested. If the resource is re-requested a ROC system has the choice of returning the same representation (possibly from a cache) or a new one. If the underlying information resource has changed since the last request, then the system *must* return a new representation. Its hard to think of a book metaphor that reflects this requirement since books are fundamentally a representation of a static resource. But consider a news website, whenever we request that site, we get an instantaneous snapshot of the news, but in ten minutes time the news will have changed and a new representation will be available at the site.

Finally we can consider the nature of physical resource representations. In an ROC system a representation may be of any physical form that the resolved provider is capable of producing. For example, the representation might be a Java object, an XML document, a Ruby object or any other suitable information structure. The requestor of the resource may indicate a preference for the form of the representation at the time of the request. If there is a mismatch between the form preferred by the requestor and the form produced by the provider then a ROC system can intermediate and attempt to find a transformation that can change the representational form. This operation is a lossless, isomorphic conversion of the resource information from one representational type to another. To distinguish from a non-isomorphic transformation (i.e. a regular computational function that produces a new and distinct information resource), this lossless operation is

called *transrepresentation* (or more commonly abbreviated to *transreption*). We can see that this is not unfamiliar in the real-world, when we perform a language translation of Jack and the Beanstalk we endeavor to create a lossless transformation. Equally, this idea has always been at the heart of computer science, we are very familiar with the idea of losslessly transforming source-code into machine executable instructions – we call this compiling. Equally when we parse a file we losslessly transform the physical representation from disk to memory.

Transreption is the isomorphic lossless transformation of one physical resource-representation to another

Finally, and perhaps most importantly for a software system, new resources may be computed by applying operations to existing resources. Operations are themselves identified within the address space and they operate on the information/resource model with semantics that are relevant to that model. The identifier for the computed resource may be expressed as a combination of the operation identifier and the resources that are operated upon to create the new resource. To use our metaphor for the last time, we can easily edit together "The Big Scary Book of Children's Fairy Tales" as a compendium containing the Jack and Beanstalk story and it can have its own ISBN. Our last axiom is then:

Computational results are resources and are identified within the address space

In summary, the principles of resource-oriented computing are:

1. A resource is an abstract set of information
2. Each resource may be identified by one or more logical identifiers
3. A logical identifier may be resolved within an information-context to a physical resource-representation
4. Computation is the reification of a resource to a physical resource-representation
5. Resource representations are immutable
6. Transreption is the isomorphic lossless transformation of one resource-representation to another
7. Computational results are resources and are identified within an address space

In the next sections of this white paper we will examine examples of existing information systems that exhibit some of the principles of ROC. We will then look at an implementation of a complete

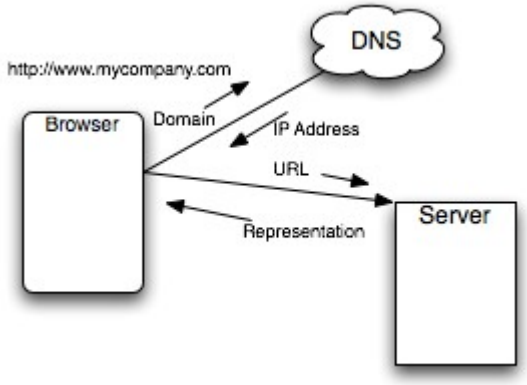
ROC system for building general software applications and systems.

Examples of Resource-Oriented Systems

In this section we will look at three examples which exhibit, in varying degrees, some of the principles of a resource-oriented computing system. The first is the World Wide Web, the second is the set of Unix command line tools, and the third is Yahoo! Pipes[2], an end user content aggregation and management tool. After examining these examples we will look at the application of resource-oriented computing to general software development.

World Wide Web

The World Wide Web is a distributed information system in which information is identified by Uniform Resource Locator (URL) logical addresses. When a resource is requested, the logical address is resolved in stages. First, the domain name portion of the URL is resolved by the Domain Name Service (DNS) into a physical Internet Protocol (IP) address. Then the whole URL is sent to that IP address for processing. The recipient of the request is a web server[3] which resolves the remainder of the address, makes a copy of the requested information, and sends back a *representation*.



Let us now examine how the World Wide Web follows some of the principles of resource-oriented computing. Following axiom #1, all information in the Web is abstract. For example, the address http://www.mycompany.com/photos/john refers to a resource (possibly a photograph) and the web server may return a representation as a PNG, JPEG, GIF, or any other image encoding or it may return an HTML page. Axiom #2 states that multiple logical identifiers may map to a single resource, this is supported by mappings within both the DNS system and the web server. Two domains may map to the same IP address and different URLs may map to the same resource within a web server. Axiom #3 concerns the resolution of a logical address to a physical resource-representation, this is the core role of a web-server, an example that is very common is when a set of URLs are mapped to a physical directory of files on a host file system. Axiom #4, that computation is the reification of a resource into a physical representation, is a formal statement which corresponds with the processing that must occur on the computer running a web server. The immutability requirement of Axiom #5 is not uniformly ob-

2 http://pipes.yahoo.com

3 Example web servers are Apache, Microsoft IIS, etc.

served in the web, but is partially observed in the caching of representations in the web-browser and proxy-servers. The formal idea of transreption as an isomorphic transformation of a representation in Axiom #6 is not found in the Web. The location of computational information in the address space presented by Axiom #7 is not a general property of the web but is sometimes used in RESTful dynamically generated web sites where a parameterized URL expresses the location of the computational resource generated in response to a web request.

Even as a partially complete resource-oriented system, the Web has very attractive properties. The lack of coupling between the client and server results in high degree of flexibility. Changes may be introduced gradually without disrupting the Web. Servers and browsers may be implemented using a wide range of technologies and may be upgraded independently at any time. Due to its innate separation of the logical resource space from physical implementations, the Web has phenomenal scalability. For example, load balancing enables a single logical URL address to be physically handled by one or many thousands of servers – this is the reason why, for example, google.com can work at all. In addition, because resource representations are immutable physical data, they may be cached under their logical identifier (URL) at any point in the network, this property minimizes the computational load of the web server and the data transfer cost to the network.

The economic properties of the Web, and its resource-oriented nature, have led to its success. In the Web, *the cost of change is less than or equal to the value added*. Whether a small participant with a single web site or a large corporation such as Yahoo! or Google, participation in the web is manageable technologically and economically.

Unix and Unix Tools

Another well known system with resource-oriented characteristics is the Unix operating system and, in particular, the POSIX tool set. These programs, such as grep, awk, sed, and others, work with a resource model of files containing line-delimited ASCII text. Unix provides an abstract tree-structured file system which implements a uniform logical address space. All resources and tools are identified with logical addressing, their filename, which may be a direct mapping to a physical inode or might be an indirect reference via a symbolic link. For example, a file resource might be identified as /home/rsk/readme.txt and tools to operate on that resource are invoked by requesting their logical identifier names in the logical address space. The physical code that is executed depends on the resolution process performed by the Unix operating system to map the logical request to the physical code. The interface between tools is the Unix pipe ("|"), an abstract file resource that moves data from the output of one tool to the input of another. For example, if the file /home/rsk/readme.txt contains:

```
Hi!
This is a readme file.
```

Then the execution of this set of pipelined commands:

```
cat /home/rsk/readme.txt | grep file
```

results in the following output where only the lines that contain "file" are retained.

```
This is a readme file.
```

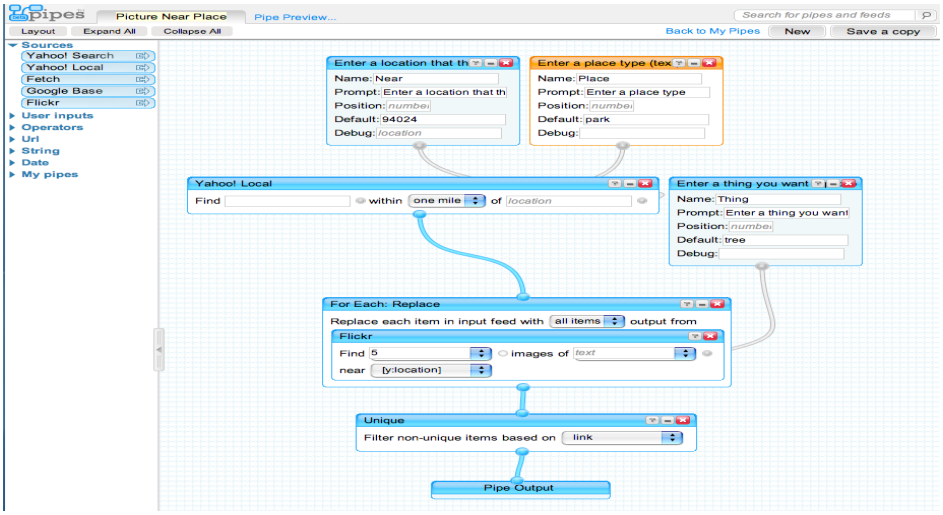
If a multi-step process or some decision making capability during the operation are required, then these pipelined requests can be orchestrated with a scripting language.

The Unix command-line tools constitute the earliest example of a computing system demonstrating aspects of resource-oriented computing. Resources are abstract binary streams identified with a logical address (based on the file system abstraction). When requested, the resources are provided as a stream of immutable information. There is only one resource type so there is no possibility to convert to another format. And, all operations, including operations qualified using modifier switches, are semantically valid when viewed within the ASCII text resource model.

It is interesting to note that, just as in the Web, the physical technology (such as programming language) used to implement each tool is irrelevant. Scripting is supported in this model – Bourne Shell scripts, Tcl scripts, and other languages can be used to orchestrate the processing of resources.

Yahoo! Pipes

Yahoo! Pipes is an end-user web feed programming tool. It uses a web browser based graphical editor to allow users to specify pipelined processing of information represented as either RSS or Atom feeds. A user can reference one or more feed resources as URL address-



es in the Fetch tool. The feed representation can then be operated on with tools such as Count, Sort, Union, Filter, and Truncate.

Yahoo! Pipes is an application that openly borrows and adopts the Unix tool abstraction to provide a composable processing system and so naturally demonstrates a subset of the resource-oriented computing seen in Unix. Information is abstract until it is requested and then representations are returned in the form of Atom or RSS feed representations. All operations are logical and function at the level of the information model.

General Software Development

Starting seven years ago, in Hewlett-Packard Laboratories and for the past five years at 1060 Research we have been pursuing the answer to a simple question *"What if resource-oriented computing was applied to general software development?"* The journey has covered more ground than we anticipated but the discoveries along the way have been both exciting and rewarding. At a high level, we have discovered the following benefits:

- ROC software inherits the economic properties seen in the World Wide Web.
- ROC permits complexity to be managed leading to overall quality gains.
- ROC software is more flexible, which dramatically increases development agility and reduces life-time maintenance costs.
- ROC software performance increases in surprising ways and scales linearly with CPU cores.
- ROC systems require less operating memory.
- ROC requires orders of magnitude less code.

The remainder of this white paper will examine a complete resource oriented computing platform and explore how its implementation provides an answer to the question we set ourselves.

A Model for General Software Development

To understand how to apply ROC to general software development we first specified a computational model and then built a concrete implementation in the form of *1060 ® NetKernel ®*. NetKernel 3.1 is a mature, robust ROC platform used in production systems ranging in scale from small (embedded), to large (displacement of J2EE type enterprise systems), to distributed (distributed food chain tracking system).

NetKernel embodies *all* of the axioms of resource-oriented computing in a uniform and self-consistent model. Let's see how

URI Addressing Details

The URI specification allows for the creation of new address schemes. The general form of a URI is:

```
{scheme-name} : {scheme-specific-address}
```

In addition to the URI address schemes that work well across computers such as http:, and schemes that work across a computer such as file:, NetKernel uses new schemes that work well within a software system such as ffcpl:, active:, var:, and others.

The ffcpl: scheme is similar to file:, identifying resources within a module.

The var: scheme identifies resources which hold values during a computational process.

The active: scheme references software operations and is essentially a functional programming language encoded as a URI. For example:

```
active:random
```

identifies a random number generator. A plus sign "+" in the active scheme is used to indicate parameters. In the following:

```
active:random+min@10+max@100
```

the text "+min@10" specifies a parameter named "min" and associates the value 10; the text "+max@100" specifies a parameter named "max" and associates a value of 100. This is analogous to a more traditional function call syntax:

```
random(10,100)
```

URI addresses can be used as the value for parameters. For example, in:

```
active:xslt+operator@ffcpl:/style.xsl
+operand@ffcpl:/data.xml
```

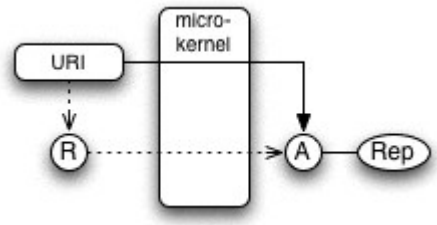
the xslt function is given references to resources for the "operator" and "operand" parameters.

these are manifested in NetKernel.

NetKernel ROC Platform

Axiom #1 states that resources are abstract sets of information. In NetKernel all resources are abstract and are not directly manipulated. Instead a resource may be logically requested and resolved to a physical *Accessor,* a software endpoint, that can access (compute) a physical resource representation.

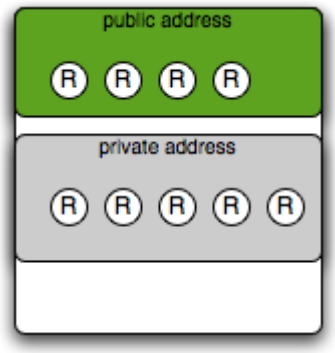
Axiom #2 states that each resource may be identified by one or more logical addresses. In NetKernel there are no exceptions - *all* resources are *logically* identified: whether the resources contain business information, configuration information, or calls on software functions to transform resources. NetKernel uses the generalized Uniform Resource Identifier (URI) as one of its standard logical identifier models (see side-bar). Uniform addressing is important for two reasons. First, indirection via a logical address ensures that all resources can be dynamically computed or substituted at run-time. Second, uniformity ensures the abstraction is self-consistent and does not suffer from corner-cases or exceptions[4].



Indirect Addressing

Unlike object-oriented programming, where objects may hold direct physical references to other objects, resources in NetKernel refer to each other by a logical URI address. NetKernel implements indirect URI addressing by directing resource requests to a micro-kernel acting as an intermediary. The micro-kernel is a URI address resolver which locates the accessor (shown as the "A" in the diagram) that can return a representation for the identified resource. In NetKernel all references, external and internal, are resolved similarly.

Address Spaces and Modules



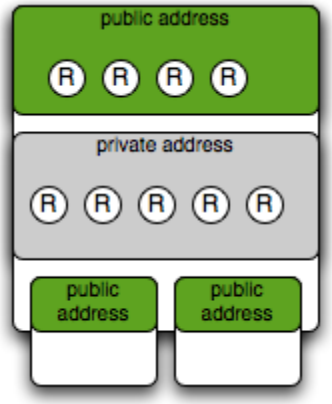
NetKernel Module

Axiom #3 states that logical identifiers are resolved to physical resource-representations within an information context. NetKernel modules contain this information context – they are a physical container for a logical resource address space. Modules are valuable as they allow developers to group resources that have similar capabilities or serve a common purpose. For example collections of static application resources, dynamic business logic functions or database query operations can be placed in separate modules making it easier to manage

ıposition of immutable representations and dynamic creation of those representations. As will be shown later, this leads to dynamic flexibility and near-static performance.

and share these capabilities. NetKernel modules include life-cycle management including live module updates, roll-back to previous configurations, and the simultaneous use of different module versions. This is possible since the coupling relationship between modules is logically defined, as required of a resource-oriented system, and is independent of physical implementation.

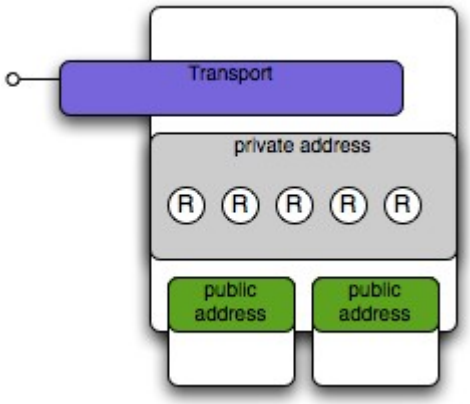
The resource address space of a module is further partitioned. All resources reside in a private internal address space. A portion of the address space can be made public and exported. Modules may import the public exported address space of other modules into their own private internal address space - applications may therefore be composed out of cleanly separable logically-linked modular units.



Module importing other modules

For example, NetKernel's mod-db module provides database connectivity capabilities. mod-db contains many Java classes, JDBC libraries, and other supporting code, all of which reside in the internal private address space. The mod-db module exports a limited portion of its private internal address space to expose access to its logical capabilities such as the function `active:sqlQuery`. Modules that require these capabilities will import mod-db and then may directly request the exposed functions.

Transports



Fulcrum Module

NetKernel supports *transports,* a mechanism that forwards resource requests originating outside the software system to a module inside and then returns a resource representation back to the external client. Transports are not part of the formal resource-oriented computing model, but are vital for building real-world systems.

A transport is a protocol handler that accepts events via protocols such as HTTP, SMTP, JMS, etc. and translates these events into resource requests that are issued into the address space of a hosting module. A NetKernel module that hosts a transport is called a *fulcrum*. A NetKernel application is usually composed of one or more fulcrum modules and the application modules which the fulcrum imports.

Computation

Axiom #4 states that computation is the reification of a resource to a physical resource-representation. Axiom #7 states that computational results are resources and are identified within an address space. In NetKernel computation can take place behind any logical address. In fact, there is an important duality in NetKernel, resources are identified by URI addresses and computations are identified by URI addresses therefore the result of all computation is considered to be a resource.

Representations and Transreption

Axiom # 5 states that resource-representations are immutable. NetKernel provides a number of resource object models for industry standard resource types such as XML and these are implemented to present immutable read-only interfaces. Application-specific resource models can be easily added to the system.

Resources are abstract in NetKernel. They are also typeless. For example the resource "the list of people attending a conference", might have the URI `ffcpl:/conference/attendance-list` and the representation might be an array of Java String objects, a spreadsheet, or an XML document fragment.

In NetKernel a resource provider may be able to provide concrete representations in a variety of formats or only one preferred format. The client requesting a resource may accept any format or it may specify a specific representation format. The relationship between the requestor and provider is not dictated by the physical type of these resources – it is only constrained by their logical relationship in the address space.

Axiom #6 defines transreption as the isomorphic lossless transformation of one representation to another. In NetKernel, all resource requests are handled through the intermediating microkernel which makes it possible to detect mismatches between the client's requested format and the provider's ability to produce the format. If there is a mismatch, NetKernel searches for a Transreptor that can perform transreption from the available to the desired representation. This late-bound logical decoupling provides enormous malleability to software, enabling change to be absorbed and accommodated with minimal cost.

Summary

NetKernel's computational model is a proven resource-oriented computing system for general software development. NetKernel has been used for a wide variety of systems including data analysis processing pipelines, cross-domain enterprise data integration, web-applications, service-oriented composition, discovery and fulfillment and even peer-to-peer autonomous network nodes

in very large distributed systems. In every case, the proven benefits of ROC seen in Unix and the Web are found to translate to general software and have resulted in dramatically simpler and more performant projects.

Architect's View

A software architect can approach resource-oriented computing as a clean, consistent, and natural way to think about and express software systems because of its primary focus on information. Architects have long aimed to develop and maintain a clean separation between the logical information model of a system and its physical implementation. Even when they are successful, system architectures degrade over time during the project cycle as changes, additions, and corrections are applied in ways that cause the physical code to deviate from the original logical model. In a ROC system the implementation is anchored at the logical level – operations applied to resources are semantically relevant at that level of abstraction. And, since code to deal with type conversions and other such details are simply not needed in an ROC system, the code that does remain is easier to read and understand.

Architects who use NetKernel to build systems report that the ROC approach dramatically reduces perceived complexity, increases their ability to rapidly build accurate and expressive systems, and reduces overall design, development and maintenance costs.

Architects working with NetKernel develop a mental model similar to the visual presentation of Yahoo! Pipes. They see information flowing through a NetKernel application just as Atom feeds flow through Pipes. The term we use is *information channels.* Just as end users can think about Atom and RSS feeds in Yahoo! Pipes without worrying about the details, NetKernel architects find that they can spend their time designing the information model for a system, the relationship between resources, the layering of their application and the flows along information channels instead of a myriad of tangential details.

Furthermore, because ROC works with sets of information, each connection and relationship established in NetKernel represents not individual datum but rather collections of data. This is important for two reasons. First, it further reduces the overall complexity of applications as there is simply less to think about. Second, computing is fundamentally based on set theory. For example, the foundation of relational databases is mathematical set theory and SQL queries are really set operations (Union, Intersection, etc.). In an object-oriented environment there is an impedance mismatch between the sets of a relational database and the points of data represented by objects – this is called 'the object-relational problem'. In a ROC system the set of information returned by

a database query can be represented naturally as a set-centric resource such as, for example, a single XML document. In NetKernel sets of information can be processed in their entirety with single instructions.

Architects will also appreciate that once resources, relationships, and channels are defined, creating a finished application is straight forward. In fact, in most cases the prototype *is* the application and no additional "development" phase is required for the project.

Here is a partial list of decisions that an architect does *not* have to make when using NetKernel:

- Object-Relational mapping product or technology. With a resource-oriented system there is no impedance mismatch with a relational database and no mapping product or technology is required.
- Caching product or technology. NetKernel is built upon an advanced caching system.
- Application server. NetKernel is a complete managed enterprise server platform. No application server product or technology is required.
- Frameworks. The decision to use J2EE, Spring, Struts, JSP, JSF, etc. is not required. NetKernel is a complete, extensible development and execution environment. No other frameworks are required.
- Compromise small size versus scalability. NetKernel scales from cell phones up to multi-core and multi-CPU computers.
- Protocol support. In NetKernel, protocols are independent of the application. Any protocol can be added later – even while the application is running.
- REST vs SOAP vs WOA. NetKernel concurrently supports all popular SOA flavours for invoking and exposing web services.

ROC Process

Architects can view NetKernel system development as comprising three stages: *construct*, *compose*, *constrain*. Construct refers to building new resource models or resource-level software functions. Frequently the Construct phase is not required as the existing body of resources and functions is rich and diverse. Compose refers to the creation of resources from lower level resources. For example, a web page might be composed of a menu resource, title resource, and the body. The application of constraints is last – this is when security, auditing, validation, or other restrictions are applied to the general software solution. As we will see later, this order is significantly different from object-oriented development in which constraints (in the form of typed classes) are applied first[5].

Construct

Construct is the first stage in the ROC programming process. The purpose of this stage is to create resource models, which include physical resource representation formats and resource-level software functions, required by an application. In most cases this stage is not actually required as NetKernel includes a rich modular library of resource models and functions.

A resource model is a collection of related representation formats and software functions. For example, NetKernel includes the industry-standard XML resource model and related functions. XML is a flexible and expressive resource model that is increasingly being used for intra and inter systems communication. NetKernel supports the representational types DOM, JDOM, SAX, and Stax, along with transreptors to convert between these types. Also included is a family of industry-standard software functions such as XSL transformation, XQuery processing, XPath support, etc. The mod-db module provides database connectivity functions that are able to process and return XML documents representing queried information.

Some application domains will require custom resource models. For example, hospital information management systems frequently use the HL7 information exchange format. In situations where a new resource model is required, the supporting code can be designed and built using Java and traditional object-oriented development techniques. Once built, the resource model becomes a part of the overall capabilities of NetKernel and the resources can be composed and constrained just like others.

Compose

Composition refers to the logical linking of a system in order to create a solution. A composite system is effectively a process which computes new resources from lower-level resources. Starting

5 This early application of constraints is one reason object-oriented code is brittle.

with the definition of the resources that an application returns to users, an architect can identify the constituent components and then define an information channel to bring them together.

Composition can be achieved in a number of flexible ways. Just as in Unix, atomic service-based tools can be linked to together as functional pipelines via URI expressions. Equally, composite and stateful orchestrations can be written in any of several dynamic scripting languages – it will be discussed below that the execution of dynamic code by language runtimes is treated in a uniform and consistent way. Finally, it is quite straightforward to create declarative descriptions for compositions which can be dynamically interpreted to fulfill the composition process.

Whichever way the composition is assembled, the resulting information process is uniformly modeled in ROC as the source of new information resources. And since all resources are logically located in the address space then it is always possible to treat the composite assembly as a black-box behind another logical address. In this way, at one layer of an application you can concentrate on the assembly of tools or resources to solve the problem – but at the next layer up, the composite can be conceptualized as a single dedicated tool or resource which itself can be incorporated into higher-order compositions.

Constrain

The final development stage is the application of constraints. Deferring the application of constraints until the end may be the biggest difference between resource-oriented development and other approaches. However, this ordering makes sense. Constraints are the way in which we enforce integrity on an information process. They should not impact the overall structure or business validity of an architectural design. Constraints are rules that provide boundaries or limits on the processing potential that already exists. A simple constraint might be a range of dates allowed in a user input field or a specification that the access to certain information resources be limited to users with specific credentials.

The application of constraints can be viewed as a layer that is applied over an existing application. In many cases in NetKernel this is literally how they are applied. For example, the application of a security policy constraint is achieved by overlaying the Gatekeeper service. This is simply a wrapper to an address space that examines supplied credentials and either forwards the request or rejects it. The Gatekeeper can be applied or removed by simply changing one line of code in an application.

Constraints can be applied interactively at the application level – for example, to enforce semantic integrity on data inputs, by, for example, raising questions such as: is this a valid username? They may also provide quality assurance within and between parts of a composite solution. Used

this way a resource validation constraint can ensure a developer has correctly implemented the specification. Validation constraints can be applied during runtime if necessary or for performance can be applied only during testing and deployment integrity checking.

The key takeaway is that constraint in ROC can be considered as separable and independent from the core composition of the information system. Perhaps surprisingly, by allowing constraint to be applied in a decoupled manner, the overall integrity of an ROC information system is actually increased relative to that possible in the constraint-first world of traditional type-focused programming.

Developer's View

For developers, ROC offers a clean and simple environment in which to craft flexible, composable computational units. With ROC's clean and clear separation of the logical information process from the physical implementation, developers can concentrate on well defined, constrained problems without having to deal with complexities leaking in from the overall system. Nor does the developer need to absorb and consider the whole of the general business process being implemented.

In addition, since an ROC system is focused on abstract information resources, physical considerations such as the choice of programming language and specific object model are secondary decisions. Systems can be composed from units written in the language whose characteristics match the task they are solving – whether that is a technical, economic or skill-set consideration.

In this section we will examine ROC from the development perspective.

Language Runtime as Service

In NetKernel the computation of resources can be accomplished in a wide choice of languages. The code for a computation may be executed by a *language runtime,* which is itself a logically located service. For example, to run a JavaScript program the following URI is used.

```
active:javascript+operator@ffcpl:/src/program.js
```

In this example, the JavaScript runtime is invoked by issuing a URI request for active:javascript and the language runtime engine is told the code to execute by providing the URI to the code as the value of the “operator” parameter. Similarly a Ruby program can be run with the following.

```
active:ruby+operator@ffcpl:/src/program.rb
```

JavaScript and Ruby are both procedural languages. Non-procedural languages and domain specific languages are also supported. For example, an XSL transform can be invoked with this URI.

```
active:xslt+operator@ffcpl:/src/style.xsl+operand@ffcpl:/src/data.xml
```

No matter which language is chosen, the purpose of all computation is to create new resources and their representations. As you can see from these three examples, the result of each computation is identified by the URI and can be used just as any other resource within NetKernel.

NetKernel provides a wealth of languages. However, if a new or specifically tailored language is

more appropriate, these can be designed, developed, and integrated into NetKernel. Examples include business processing languages, templating languages, or any new popular scripting language.

Development Process

Developers will find that they will work on one or more of the essential NetKernel development stages: construct, compose, constrain.

Construct

The purpose of the Construct stage is to create new resource models and supporting functions. Often this is not required because the existing resource models are sufficient. If required, new Accessors are written to serve as resource request endpoints. Accessors can be written in any scripting language, or may be developed in Java – in either case, the task is made painless by a uniform and consistent model and development API. Transreptors and fundamental resource models are coded in Java in the current NetKernel 3 series of products[6].

In the Construct phase developers will work with the NetKernel Foundation API (NKF). This API provides a way to issue and accept resource requests, to generate a representation, and then to send that representation back as the response. Whether the endpoint is an accessor generating a new representation or a transport connecting with an external system, the NKF API is consistent and straightforward.

Compose

The majority of development time will be spent in the compose stage. Composition is focused on the creation of new resources from constituent resources. Composition can be done in a number of different ways. New resources can be created using templating (e.g. XML Recursion Language - acts as a templating language for XML), functional composition (URIs can be chained together using functional programming with the active: URI scheme), or Unix-like scripting of sequential processes.

Since all resources are logically referenced by URI, developers can adopt techniques previously seen outside of software development, such as the use of scaffolding. It is a simple matter to start system development with a collection of static resources and use these to compose the broad system together, layer by layer. For example, the address ffcpl:/index.html could be linked to a static resource located at ffcpl:/src/index.html while the front-end of the application is constructed.

6 The next version of NetKernel will support an expanded role for scripting languages in this area.

```
<link>
 <ext>ffcpl:/index.html</ext>
 <int>ffcpl:/src/index.html</int>
<link>
```

Later the same URI can be linked to code that will dynamically create the resource:

```
<link>
 <ext>ffcpl:/index.html</ext>
 <int>active:xslt+operator@ffcpl:/src/style.xsl
+operand@ffcpl:/src/index.xml</int>
<link>
```

Composition therefore allows the construction of robust logical relationships within the software system and allows the details of the information itself to be treated separately from the process.

In addition, should the system require changes, the logical relationships present in the composite system can be readily modified and the results tested in a live running system. The immediacy of dynamically linked and evaluated development is a very significant productivity enhancement as developers can retain the context of the development process in their head while experiencing instant feedback.

Constrain

The application of constraints is the last stage of the development process. In this stage various constraints can be layered on top of an existing, working system. A developer can use overlayed constraints to assist with overall project delivery – a functional unit of a system can be validated using test resources. Validation constraints can provide proof of completion and also can later be used to provide long-term system quality assurance.

In addition, it can be very useful for a project team to choose to constrain the palette of development tools that they will use to deliver their solution. This can be accomplished by constructing a module that imports the desired selection of tools from other libraries and only exports a subset of specific interest to the project. This new module can then become the standard toolbox for the application.

Performance

In the discussion so far, the broad theme of resource-oriented computing is the transition from static physically linked code to dynamic logically linked information processes. On first consideration one might expect a performance overhead. However, and counter-intuitively, it is consistently found that ROC actually offers significant performance advantages. In this section we will explore computational performance of ROC.

Physical Execution Decoupling

Resource requests are issued to the kernel which, acting as an intermediary, resolves the address to a physical accessor endpoint and then invokes the accessor. When the accessor completes its task, it returns the physical resource representation to the kernel which forwards it to the initial requestor. Since requests are not coupled to the physical execution of the code in the endpoints, the kernel is free to manage the assignment of each computational task to a low-level thread of execution. And, since the kernel is mediating all computation, it can use its global knowledge of the system to manage any individual process to ensure optimum efficiency of the whole software system.

At the logical ROC level there are no threads, only requests for resources – the logical system is inherently asynchronous. A thread must perform the computation but the kernel is free to use an optimal strategy to determine which thread to use and whether to use asynchronous or synchronous dispatch on the thread. This flexibility allows for an optimization that minimizes computationally wasteful context switching in the CPU and ensures 100% thread utilization can be achieved as there is no need to waste computing resources by blocking threads.

An additional performance payback of logical indirection is in the scaling and optimal utilization of SMP or multi-core processor architectures. In ROC a software function has a single logical location but the assignment of the processing to threads occurs once the function has been requested. This is directly analogous to the near linear scaling that load balancing offers in Web-server architectures. In this case, ROC enables load balancing to be brought down to the finest granularity of software and the execution of physical CPU threads.

Finally, the complexity and system-integrity challenges of developing thread-safe code is removed from the developer. For example, a new low-level piece of code and its associated OO library will by default be marked as UNSAFE_FOR_CONCURRENT_USE. The system architect integrating the new functionality can be confident that the system is safe to use immediately, knowing that the new code will never be scheduled concurrently. After review and testing, this declarative con-

straint can be removed and the kernel will seamlessly parallelize and optimize the execution of the code.

Caching

In ROC every computational result is a resource and has an identity in the address space. This seemingly simple statement leads to dramatic performance implications. Since the outcome of every computation is identifiable then it can be cached under that identity. If the same resource is requested again then it can be looked up in the cache rather than execute some physical code. It is well known in computation theory that looking things up is much much cheaper than working them out.[7]

In an ROC system, such as NetKernel, every computational result can be cached. The kernel can manage the cache using an evolutionary 'survival of the fittest algorithm' – balancing memory use by discarding those resources that are least valuable (i.e. those that have not been reused, or are just very cheap to recompute). Interestingly, any software system is dynamic and the set of useful resources changes constantly. The NetKernel cache therefore achieves a dynamic equilibrium and retains, ready for immediate use, the optimal possible subset of computational resources. In general business systems, we find that at least 30% of information has a lifetime longer than a single transaction and so has the potential for reuse without recomputation. In the case of long lived dynamic systems, having long periods of pseudo-static operation, this can rise to close to 100%. Perhaps the most interesting observation of ROC caching is that there is no need to anticipate what the valuable information is ahead of time. An ROC system naturally self-balances around the sweet spot.

7 Dictionary lookup is logarithmic whereas computing a resource is generally linear or worse.

Object-Oriented Computing Comparison

How does resource-oriented computing compare with other approaches, in particular object-oriented programming?

A side by side comparison of ROC and OO is not really the right perspective as ROC and OO are not mutually exclusive. Indeed they support and augment each other. To visualize the relationship, think of ROC as a logical level resting on top of a physical OO foundation. ROC is fully anchored on a logical computing level – resources are abstract, identifiers are logical, etc. while OO is anchored on a physical computing level – objects are concrete in memory and object references are physical memory pointers. Computations must ultimately be performed at the physical level, so it is easy to see that a logical model such as ROC must be implemented with, and must run on, a physical platform. In the case of NetKernel, the physical kernel is implemented in Java[8].

Another way to visualize the relationship is to see that ROC is an independent, logical computing model. ROC is not an extension of OO nor is it a framework for OO. There is no "edge" to the ROC model where it "runs out of juice" and reverts to a lower level model. An ROC implementation doesn't even need OO as it could be written on a non-object-oriented platform. However, OO does provide some powerful capabilities and abstractions that makes it a preferred foundation for ROC.

The strengths of OO and the weaknesses of particular implementations (e.g. C++, C#, Java, etc.) are well understood. In general, OO serves well for e.g. small units of business logic, but suffers when those units are linked to create scaled up solutions. The fundamental scaling problem lies in the brittle nature of OO code with respect to changes in business information. On a small scale, when business information is changed then code changes ripple through the class hierarchy. Some of the impact can be managed with tools such as IDE refactoring. However, on a larger scale, change occurs across the boundaries of the logical application model, at the semantic and structural foundations; beyond the scope of the physical refactoring of an IDE and outside an individual developer's responsibility. Such systemic changes in OO systems dramatically increase the cost and risk of change. Even on a small scale, changes to an OO system, at a minimum, require a recompile, test, deploy, restart, and application state reload .

In ROC change can be continuously managed. With ROC information is abstract and typeless. In fact, information, operations and transformation and all aspects of a program's structure and rules are logically related. As we have seen from the axioms of ROC, binding to the physical computing level is deferred to the moment of a resource request – and once that is complete, every-

8 There is nothing special about the use of Java for NetKernel. NetKernel could be implemented in other languages and platforms.

thing is back at the logical level. What is apparent from analysis and practice using ROC is that it accomplishes several important things. By treating information as abstract and locating it logically within an address space, it eliminates the brittleness found in OO coding. The majority of investment in application design and development is made at the logical level – at this level it is easy to reconfigure an application to respond to changes to business information and is even robust in the face of significant changes at the physical level.

Note that ROC is *not* an example of a model being used to generate code, which would still result in mountains of brittle code, instead ROC is a logical model that is executed in real-time on top of a physical computing level. The run-time flexibility this provides is just as important as the design-time flexibility. At run-time each logical reference for resources or services is resolved for each request and since the model is typeless, the physical implementations can be changed *while* a system is running without disturbing the application.

If ROC introduces a new opportunity to maximize system performance *and* solves the problem of brittleness found in OO code for larger scale systems, what is the minimum size of system that can gain from ROC? It doesn't make sense to construct primitive algorithmic functions with ROC as this construct-phase[9] task essentially extends and enhances the physical computing level supporting ROC. However, as soon as relationships between logical parts must be made, these are best done at the logical level with ROC.

9 Some low-level algorithms perform better coded at the ROC logical level. For example, the Fibonacci double-recursion algorithm has a linear cost at the logical level and an exponential cost at the physical level. Finding similar results with other algorithms is an active area of research.

Summary

Resource-oriented computing is a new approach with a long history, and is grounded on a set of fundamental principles combined into a simple self-consistent model. The model resides in a logical computing level fully divorced from the physical computing level. The clean separation of logical and physical results in simpler and more flexible programs *and* faster, more easily scaled systems.

The benefits of using resource-oriented computing are not theoretical. Empirical evidence from enterprise adopters of NetKernel consistently report the following:

- Development time is measured in weeks, not months and years.
- Code required is 10x to 100x less than other approaches.
- Life-cycle costs are dramatically less because of inherent flexibility.
- Systems typically run faster by a factor of three to four compared to Java J2EE.
- Systems scale linearly with CPUs and cores.

1060 NetKernel release 3.1 is a mature, robust realization of the resource-oriented computing model that supports software development from small scale systems to large enterprise and distributed systems.

Learning to develop software systems using ROC is not difficult, just *different*. It takes less time to learn to use NetKernel than it does to learn J2EE. The return on learning the new approach is considerable but, even more importantly, it's also a lot of fun!

For additional information about Resource-Oriented Computing, including ROC architectural consulting services and training, please contact 1060 Research at www.1060research.com