

1060 Research Limited

Collection of Articles

Resource-Oriented Computing[®] and NetKernel[®] 5

v 1.1.1



1060, NetKernel, Resource Oriented Computing, ROC are respectively registered trademark and trademarks of 1060 Research Limited

2001-2013 © 1060 Research Limited

Simplicity is the ultimate sophistication.

[Leonardo da Vinci](#)

Most software today is very much like an Egyptian pyramid with millions of bricks piled on top of each other, with no structural integrity, but just done by brute force and thousands of slaves.

[Dr. Alan Kay](#)
[\(father of OO programming\)](#)



This software transformation, whatever it is, is coming. It must come; because we simply cannot keep piling complexity upon complexity. We need some new organizing principle that revamps the very foundations of the way we think about software and opens up vistas that will take us into the 22nd century.

[Robert Cecil Martin, aka "Uncle Bob"](#)

In order to handle large systems over time, there needs to be a high level description of information...a language which allows the uniform representation of information. Algorithms is one type of information that has to do with computations, but there are other types.

...Computations need not be arranged in programs; they must be information, standing on their own, allowing the user to mix them at will, just like with any other type of information.

[Terry Winograd](#)

Table of Contents

<u>A Brief History of Netkernel</u>	4
docs.netkernel.org	
<u>NetKernel Introduction</u>	7
Randy Kahle at InfoQ	
<u>Decoupling REST URLs from Code Using NetKernel Grammars</u>	18
Randy Kahle at InfoQ	
<u>A RESTful ESB implemented using NetKernel</u>	25
Jeremy Deane at InfoQ	
<u>Resource-Oriented Analysis and Design</u>	
Peter Rodgers, PhD	
 <u>Part 1</u>	36
NetKernel News, Volume 3, Issue 34	
<u>Part 2</u>	43
Netkernel News, Volume 3, Issue 36	
<u>Part 3</u>	56
NetKernel News, Volume 3, Issue 36	
<u>Part 4</u>	63
Netkernel News, Volume 3, Issue 37	
<u>Part 5</u>	81
NetKernel News, Volume 3, Issue 38	
<u>Part 6</u>	94
NetKernel News, Volume 3, Issue 39	
<u>Part 7</u>	102
NetKernel News, Volume 3, Issue 40	
<u>Part 8</u>	114
NetKernel News, Volume 3, Issue 41	
<u>Part 9</u>	119
NetKernel News, Volume 3, Issue 42	
 <u>Resource-Oriented and Programming Languages</u>	
Peter Rodgers, PhD	
 <u>1 – Introduction</u>	125
NetKernel News, Volume 2, Issue 4	
<u>2 – Imports and Extrinsic Functions</u>	131
NetKernel News, Volume 2, Issue 5	

<u>3 – Arguments</u>	138
NetKernel News, Volume 2, Issue 7	
<u>4 – What is Code? What is a Function?</u>	148
NetKernel News, Volume 2 , Issue 8	
<u>5 – Functions, Sets and RoC</u>	158
NetKernel News, Volume 2 , Issue 9	
<u>6 – ROC: State of the Union</u>	165
NetKernel News, Volume 2 , Issue 10	
 <u>ROCing the Cloud</u>	174
NetKernel News, Volume 2 , Issue 37	
Peter Rodgers, PhD	
<u>ROC and data structures</u>	186
NetKernel News, Volume 3, Issue 13	
Peter Rodgers, PhD	
<u>On Metadata</u>	199
NetKernel News, Volume 2, Issue 33	
Peter Rodgers, PhD	
<u>On State and Consistency</u>	219
NetKernel News, Volume 2, Issue 1	
Peter Rodgers, PhD	
<u>DPML – Declarative Markup Language</u>	224
Source: docs.netkernel.org	
<u>DPML – A very unusual language</u>	225
NetKernel News, Volume 3, Issue 26	
Peter Rodgers, PhD	
<u>nCoDE - Compositional Development Environment with a Language Runtime</u>	230
docs.netkernel.org	
<u>Separating Architecture from Code</u>	232
Tony Butterfield, Durable Scope	
<u>Caching: How and why it works</u>	235
Tony Butterfield, Durable Scope	
<u>Visualizer – Time Machine Debugging</u>	238
Tony Butterfield, Durable Scope	
<u>Ending the Absurdity</u>	243
Tony Butterfield, Durable Scope	
<u>ROC Concept Maps</u>	246
Tony Butterfield, Durable Scope	
<u>The future is coming on</u>	252
Tony Butterfield, Durable Scope	
<u>ROC: Step away from the Turing tape</u>	255
NetKernel News, Volume 2, Issue 1	
Peter Rodgers, PhD	

How do you prove the value of ROC?

259

NetKernel News, Volume 1, Issue 23

Peter Rodgers, PhD

On Empiricism

NetKernel News, Volume 4, Issue 18, 19, 20

262

Peter Rodgers, PhD

NetKernel Newsletter Archive

- <http://wiki.netkernel.org/wink/wiki/NetKernel/News/index>

The NetKernel Newsletter is published weekly and contains news of product updates as well as a continuing series of in-depth articles on NetKernel and Resource Oriented Computing.

The latest copy is always available at the following link:

- <http://wiki.netkernel.org/wink/wiki/NetKernel/News/latest>

Video Library (YouTube)

Comprehensive Training Videos available inside the training portal. Registration required.

- [Download and Installation](#)
- [First Module](#)
- [Java Module](#)
- [Visualizer](#)

NetKernel ROC Resolution

- [One Endpoint](#)
- [Two Endpoints](#)
- [Import](#)
- [Resolution Order](#)
- [SubRequests](#)
- [Superstack](#)
- [Mapping / Aliasing](#)
- [Mapper / Superstack](#)
- [Mapping / Deep Superstack](#)
- [Limiter](#)
- [Limiter / Superstack](#)
- [Private](#)
- [Private Import](#)

Production Package Deployment

- [Part 1](#)
- [Part 2](#)
- [Part 3](#)
- [Part 4](#)

nCoDE

- [Part 1](#)
- [Part 2](#)
- [Part 3](#)

Space Explorer II

- [Part 1](#)
- [Part 2](#)
- [Part 3](#)

Golden Thread Discussion

- [Part 1](#)
- [Part 2](#)

NetKernel Grammar

- [Part 1](#)
- [Part 2](#)

NetKernel 5

The World's 1st Uniform Resource Engine

ROC Microkernel
Heatmap
NK Protocol / Cloud Infrastructure
Visualizer Time-machine debugger
Resource Resolution and Classloader Trade Tools
L2 Cache
Load Balancer
Endpoint Statistical Profiling
Encryption Modules
Real-time System Status Reports and Charts
Apposite Package Manager
xUnit Test Framework
Multit-mode deadlock detector
Role-based System Administration Security
DPML Declarative Request Language
Historical Log Analyzer
Multilevel Log Management
nCoDE Visual Composition Environment
eMail client / server
XMPP Transport
SOAP Transport
REST Web Services client / server
Relational Database Too Set
SSH daemon
Declarative Architectural Suite
oAuth client / server
High-performance Asynchronous HTTP client / server
HTTP Virtual Hosting Manager
State Machine Runtime
Groovy, Ruby, Python, Java, JavaScript, BeanShell

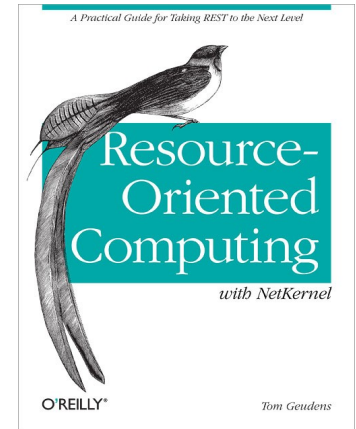
www.1060research.com

Books on Resource-Oriented Computing and NetKernel

- **Resource-Oriented Computing with NetKernel.**

Tom Geudens

O'Reilly, 2012.



- **Resource-Oriented Architecture for Webs of Data.**

Brian Sletten

Morgan and Claypool Publishers, 2013

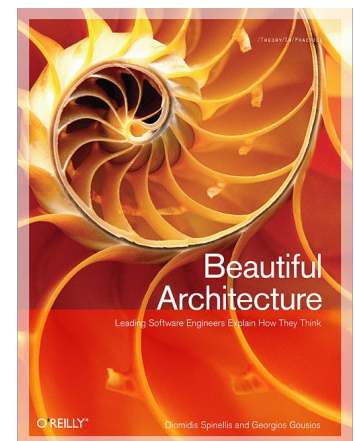


- **Beautiful Architecture, Leading Thinkers Reveal the Hidden Beauty in Software Design.**

Diomidis Spinellis , Georgios Gousios

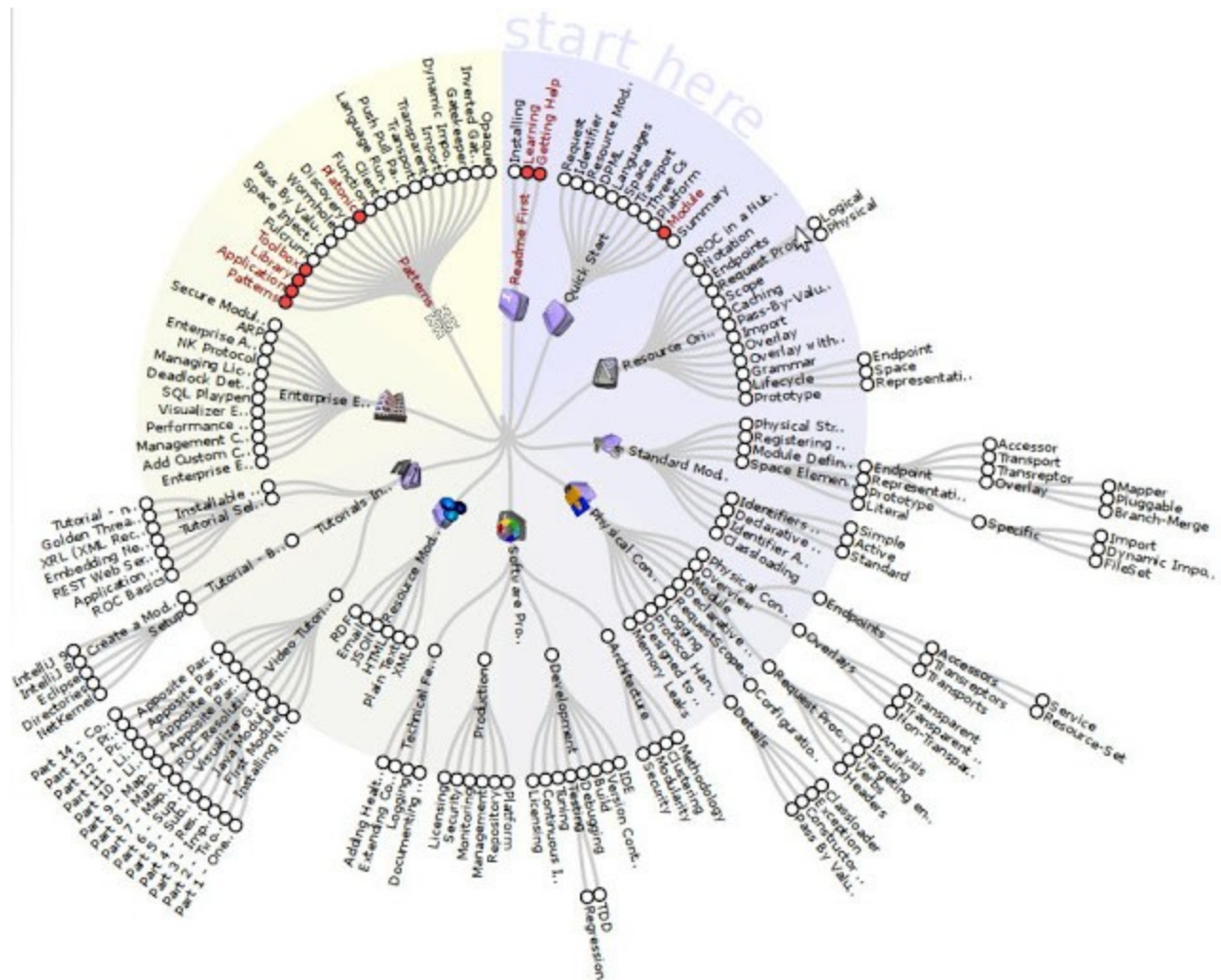
O'Reilly, 2009

(chapter 5) Resource-Oriented Architectures: Being “In the Web”



MindMap of NetKernel Universe

As part of our ongoing efforts to at least provide crampons if not materially reduce the learning curve required to reach the epiphany of Mt NetKernel I want to present our latest efforts with a visual representation of the structure of knowledge within the NetKernel documentation.



[Click here for larger image](#)

It is a mindmap; a visual map of the structure of the documentation contained in the core books. It is dynamically generated and clicking on the nodes takes you straight to the actual page within the documentation. Mindmaps are incredibly useful to see the lie of the land or the outline of available knowledge. However the real benefit I see is that various data visualisations can be overlayed on this map which will help in navigating. I have heard from many people that it's easy to get lost and end up going round in circles trying to find meaning. (Of course some of that indicates that the content needs refining and that is happening too)

The simplest visualisation is what a web browser does to highlight visited links. So in the background the documentation system locally captures analytics (hit counts and last viewed time.) This quickly allows somebody using the documentation to learn which paths have been traversed and what still remains unexplored. This tool will also contain a number of other visualisations such as pages viewed over specific time periods or pages viewed most frequently.

A Brief History of NetKernel

NetKernel is different.

To give you a picture of how it is different here's a short story...

A computer program defines functions, invokes functions, receives values, assigns those values to variables (state) and invokes more functions with that state. There are dozens of programming languages, you choose your language, you write your program, you run your program. That's how it is, that's how it has always been.

Step forward to the early 1990's. The World Wide Web was developed as an efficient distributed information system. To begin with it was static, serving files, but soon it became dynamic - when your browser requested a URL the Web server would dynamically compute the state and return it to you.

The Web is a software system that has completely decoupled the state from the implementing functions. Nobody cares if you write your web application in PHP or Java or ... etc etc etc. The only thing that matters is the name of the resource (its URL identifier) and the state of the resource when you request it.

So what?

So what? So, the Web can be instantly updated, the Web scales, the Web absorbs change, the Web is malleable and flexible. These are properties that every well run business would like to share.

So in the early 2000's people started to think this was "a good thing". People started to talk about "Web-Services". There were some absolutely abominable technologies that got labelled with the "Web" label. Fortunately many of them died and over time people have started to realize that these weren't very web-like at all. Today there is growing interest in REST.

REST is just another name for "the properties of the Web".

End of story. Or is it...

So we can expose dynamic business information and business processes (business state) using the Web. But the dynamic information is generated with computer programs that call

functions, receive values, invoke more functions etc. Because that's a "computer program" and that's how it has always been.

Not necessarily...

NetKernel and Resource-Oriented

Software developers build applications and systems with NetKernel by making requests for information resources and receive back representations of the state of those resources.

If you want a very simple analogy for how NetKernel works, it takes the core principles of the Web and applies them internally to the fundamentals of software.

Crazy Idea

It turns out that when you go down this path you acquire the Web's properties of scalability, maleability and assimilation of change inside the fine granularity of your software. But something else happens too.

A sensible software engineer would listen to this story and consider these ideas and would immediately dismiss them. "You might get architectural decoupling but the performance is going to suck". Yes you would think so, but that is to miss one very significant property of the Web: *State can be cached since it has a resource identifier*.

More Flexible and Faster

For the computer scientist, in NetKernel's resource oriented computing abstraction all functions have extrinsic memoization. Referential integrity is externally managed by the computation environment.

In resource oriented computing every request that invokes a function has a resource identifier. All state, including all intermediate state, can be cached.

The really interesting thing is that the real world obeys natural statistical distributions. It turns out that more often than not you frequently need the state of the same set of resources. Even inside low-level code. So there is a net value in holding onto state.

Completely counter intuitively, it turns out that decoupling the internals of software actually makes it faster!

At what cost?

I've heard too many cynical marketing stories about the benefits of this technology or that technology.

We all know that adopting a new technology comes at a cost. You have to learn it and you hope that the effort in learning it is going to pay you back.

As encouragement, there's one more property of building software this way. Stepping away from the code and thinking in terms of resources is scale invariant - that is, you can compose higher-order resources from lower-order resources (in the Web this is called a "mashup" - but that's a hideous expression!).

You can apply transforms to resources to create new resources which themselves can be composed.

When you apply ROC to real world software engineering, the scale of problems that you are able to solve goes up but the complexity remains finite. Its like moving from being first violinist in the orchestra to being the conductor.

Research History

Resource oriented computing has a solid historical and theoretical foundation. As described in the ROC Nutshell document:

“It can be regarded as a generalization of the World Wide Web's REST architecture and the foundation concepts of operating systems such as Unix. But with an eye to history one sees that it traces its origins back even further to Turing and his seminal papers on the Turing and Oracle-machines (Turing 1939) but even beyond that it ultimately rests on the core mathematics of set-theory and Gödel's world changing thesis on incompleteness and computability.”

NetKernel started as a project in HP labs in 1999 to answer fundamental questions about software systems. It became a product of 1060 Research, Ltd. in 2002. When NetKernel 3 was released in 2005, work began on an entirely new kernel and a simplified and more powerful abstraction. The result of these four years of work was NetKernel 4. Building upon this foundation NetKernel 5 integrates over three years of production refinements to the libraries and tools but also provides a new set of optimisations to the core microkernel and foundations. ♠

NetKernel Introduction

Randy Kahle

NetKernel is a software system that combines the fundamental properties of REST and Unix into a powerful abstraction called resource oriented computing (ROC). The core of resource oriented computing is the separation of logical requests for information (resources) from the physical mechanism (code) which delivers it. Applications built using ROC have proven to be small, simple, flexible and require less code compared to other approaches.

Hello World!

In a resource oriented system requests are made for resources and concrete, immutable resource representations are returned. This follows directly from the principles of REST and the World Wide Web. In the web a request for a resource such as <http://www.1060research.com> is resolved to an endpoint IP address by the Domain Name Service (DNS). The request is sent to that IP address and a web server delivers a response containing a concrete, immutable representation of the resource. Similarly, in NetKernel a resource request is resolved to an endpoint called an Accessor which is responsible for returning a concrete, immutable representation. NetKernel is implemented in Java so we will use this language for our first examples, however, later we will show that Java is one of many supported languages.

A request is delivered to the resolved Accessor's `processRequest` method. The following example creates an immutable `StringAspect` that contains the "Hello World" message and returns it in the response:

```
public void processRequest(INKFConvenienceHelper context) throws Exception
{
    IURAspect aspect = new StringAspect("Hello World");
    INKFResponse response = context.createResponseFrom(aspect);
    context.setResponse(response);
}
```

The context object is an interface to the microkernel which allows an endpoint to bridge between logical requests and physical code. The `processRequest` method is triggered when a logical request's URI is resolved to this endpoint. Unlike a Java Servlet where a URI request can only come from the Web, URI requests that resolve to NetKernel accessors can come

from anywhere including other endpoints.

If a Servlet needs additional information it will use references to other Java objects, creating deep call stacks, for example when using JDBC access to a database. This is where ROC begins and the Web ends. **Accessors** do not have memory references to other accessors. To obtain additional information or call other software services, they issue sub-requests up into the logical resource address space.

In NetKernel resources are identified by a URI address, again just like in the World Wide Web.

Now we leave the physical level of Java objects and see how a URI address is defined and can be dynamically bound to our code. NetKernel is a modular architecture. Software resources can be packaged in physical containers called **modules**. As well as being a physical package for deployment, a module defines the logical address space for resources and their relationship to physical code. A module is like a completely self contained micro-world-wide-web for software. The following entry in a module definition maps the logical address "ffcpl:/helloworld" to our accessor, the HelloWorld object.

```
<ura>
  <match>ffcpl:/helloworld</match>
  <class>org.ten60.netkernel.tutorial.HelloWorld</class>
</ura>
```

The binding between a request and an endpoint occurs only at the moment the request is resolved. Once the accessor completes, the relationship is decoupled. This is different from Java and other languages where bindings, once made, are permanent. This dynamic logical binding results in very flexible systems which can be reconfigured at runtime.

In the logical address space we have new degrees of freedom. We can map requests from logical to logical, logical to physical (as above) or from one address space to another. The following rewrite rule illustrates that we can use a regular expression match to change a request's URI to another: The effect of this rule is to map a request such as "ffcpl:/tutorial/helloworld" to "ffcpl:/helloworld".

```
<rewrite>
  <from>ffcpl:/tutorial/(.*)</from>
  <to>ffcpl:/$1</to>
</rewrite>
```

A module is an encapsulated private logical address space. A module may export a public address space which can then be imported by other modules. In our example, our module exports the public address space "ffcpl:/tutorial/.*", promising to provide all re-

sources located below the tutorial path.

```
<export>
  <uri>ffcpl:/tutorial/(.*)</uri>
</export>
```

So far we have not considered where a request comes from. Without an initial starting point nothing would happen in the system. A transport is an endpoint that detects (external) events. When a **transport** detects an event it creates and issues a root request into the logical address space and waits for an endpoint to be located, bound, scheduled for processing and return a representation. A module may host any number of transports and each transport will inject its root requests into the hosting module. For example, the HTTP transport detects the arrival of an HTTP request and then creates and issues a corresponding internal NetKernel root request. The following diagram illustrates the path by which an HTTP request turns into a NetKernel request and travels to our HelloWorld accessor class.

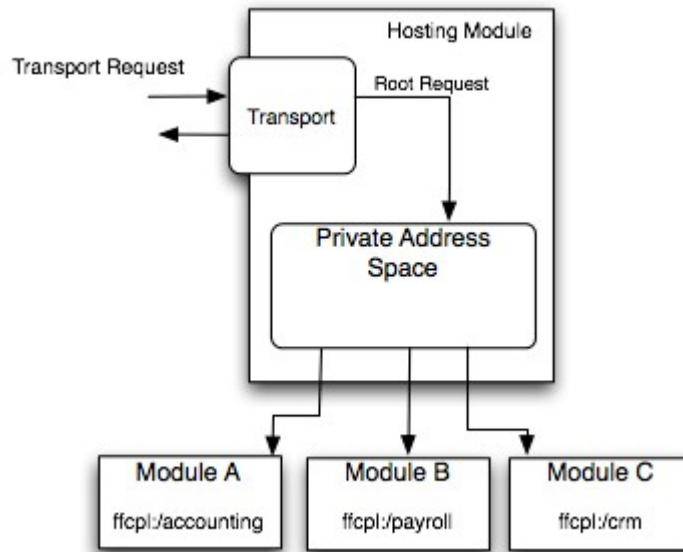
```
http://localhost:8080/tutorial/helloworld    (at the transport)
  |
  v
ffcpl:/tutorial/helloworld                    (request issued by transport)
  |
  v
ffcpl:/tutorial/helloworld                    (request that enters our module)
  |
  v
ffcpl:/helloworld                            (after rewrite rule)
  |
  v
org.ten60.netkernel.tutorial.HelloWorld      (Physical Accessor code)
```

Nothing in the system dictates how requests for our Hello World resource originate. Its address space can be simultaneously connected to HTTP, JMS, SMTP, LDAP, ... you name it - external application protocols, or even other layers of our software architecture. Each module in a NetKernel system is a self-contained encapsulated resource oriented sub-system.

Addresses and Address Spaces

In NetKernel a module defines a private address space. Only three things can occur in the private address space:

- Map a logical address to logical addresses
- Map a logical address to physical endpoint
- Import a logical address space from another module



With these three primary relations it becomes possible to craft cleanly layered and channeled architectures. Since these relationships are dynamic the architecture itself is also dynamically composable. The diagram below shows a high level view of a typical NetKernel application. A transport issues root requests into the private address space of a hosting module and NetKernel searches for an endpoint that will process the request. In our example the hosting module imports modules "A", "B" and "C". Each adds their public exported address space to the host module's private address space, specifically `ffcpl:/accounting/.*`, `ffcpl:/payroll/.*` and `ffcpl:/crm/.*`.

If in our example a root request is issued for the URI `ffcpl:/crm/contacts` it will match the exported address space of module "C" and the request will be sent to that module, eventually being resolved to physical code which can fulfill the request for `ffcpl:/crm/contacts`, perhaps by using physical objects such as a JDBC connection or more commonly by issuing a sub-request for a logical relational database service available within the scope of module "C".

Accessors

Next we switch back to the physical level and take a closer look at accessors. As we have seen, accessors are endpoints that return resource representations. Accessors themselves are very simple. They can only do four things:

1. Interpret what resource is being requested (i.e. by inspecting the initiating request)
2. Create and issue sub-requests for additional information (as synchronous or asynchronous requests)
3. Create value by performing their service - doing something!

4. Create and return an immutable physical resource representation

An accessor discovers from its context what it is being asked to do. The request URI can be retrieved as well as the argument values of named parameters. An accessor may need to know which URI was used by the current request if multiple addresses are mapped to a single endpoint. With logical / physical separation one piece of code may have multiple logical locations mapped to it.

Services are called with named parameters using the active: URI scheme. The active scheme takes the form

```
active:{service-name}+{parameter-name}@{uri-address}
```

The following BeanShell script implements the `toUpper` service. It retrieves the immutable aspect for the "operand" resource using the `sourceAspect` method with the URI `this:param:operand`. We could use the context object to obtain the calling request, look for the named parameter "operand", obtain its URI and issue a sub-request for that resource. Instead the Netkernel Foundation API provides a local internal logical address space for the arguments of a request. By requesting the URI `this:param:operand` we are effectively asking to dereference the operand pointer.

```
import org.ten60.netkernel.layer1.representation.*;
import com.ten60.netkernel.urii.aspect.*;
void main()
{
    sa=context.sourceAspect("this:param:operand", IAspectString.class);
    s=sa.getString();
    sa=new StringAspect(s.toUpperCase());
    resp=context.createResponseFrom(sa);
    resp.setMimeType("text/plain");
    context.setResponse(resp);
}
```

The script specifies that it wants the operand resource returned as an implementation of the `IAspectString` interface. However, at the logical level, code is not aware of physical level types. This leads to a new concept called **transrepresentation**. If a client requests a representation type that an endpoint does not provide then the microkernel can intermediate. When a mismatch is detected, the microkernel searches for a **Transreptor** that can convert from one type to the other.

Transreptors turn out to be very useful. Conceptually, a transreptor converts information from one physical form to another. This covers a significant amount of computer processing including:

- Object type transformation
- Object structure transformation
- Parsing
- Compiling
- Serializing

The key point is that this is a lossless transformation, **information** is preserved while the **physical representation** is changed. Transreptors help reduce complexity by hiding physical level details from the logical level allowing developers to focus on what's important - information. For example, a service such as `active:xslt` requests information as a DOM and the developer working at the logical level provides a resource reference whose representation at the physical level is a text file containing XML. NetKernel will automatically search for a transreptor that can transrept (parse) the textual XML representation into the DOM representation. The architectural and design significance of transreption is type decoupling and increased application and system flexibility.

In addition, transreption allows the system to move information from inefficient forms into efficiently processable forms, for example, source code to byte code. These transitions occur frequently but only require a one-time conversion cost and thereafter can be obtained in the efficient form. In a formal sense, transreption removes entropy from resources.

Resource Models

We have seen how a logical level URI address is resolved to a physical level endpoint and bound to it for the time it takes to process. We have seen that physical level concerns such as type can be isolated in the physical level. We also have seen that services can be called with named parameters, all encoded as URI addresses.

This leads to the idea of a **resource model**, a collection of physical resource representation types (object models) and associated services (accessors) that together provide a toolset around a particular form of information, for example, binary streams, XML documents, RDF graphs, SQL statements and result sets, images, JMS messages, SOAP messages, etc. The idea of a resource model allows a developer to build composite applications out of one or several resource models in combination, echoing the Unix philosophy of having special reusable tools rapidly combined together to create solutions.

The **Image** resource model includes services such as `imageCrop`, `imageRotate`, `imageDither` and more. Using the image resource model a developer can create image processing pipelines, all with simple requests such as:

```
active:imageCrop+operator@ffcpl:/crop.xml+operand@http://1060research.com/im-
```

```
ages/logo.png
```

NetKernel's XML resource model includes transformation languages, several validation languages and many other XML technologies. Above this, a specialization of the XML resource model is the PiNKY feed processing toolkit that supports ATOM, RSS, and many simple feed operations and is 100% downward compatible with the XML resource model. With transreptors, a developer need not know if an XML resource is physically a DOM, SAX stream or one of the several possible representation types. Using the XML resource model developers can quickly build XML processing systems. For example the following request uses the XSLT service to transform the resource `ffcpl:/data.xml` with the style sheet resource `ffcpl:/style.xsl`:

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

Sequencing

Resource request URIs are essentially "op-codes" for the resource oriented computing model. Just like Java byte-codes, they are generally too low level and would be difficult to code manually. Instead one can use a number of scripting languages to define and issue these requests. The context object we saw earlier is an example of a uniform POSIX like abstraction around the microkernel called the NetKernel Foundation API. This API is available to any supported dynamic procedural languages. In addition, specialist declarative languages are provided whose purpose is solely to define and issue source requests.

One such scripting language is DPML, a simple language that uses an XML syntax. Why XML syntax? Because in a dynamic loosely coupled system where code is a resource like any other it is very straight forward to create processes that dynamically generate code. And XML syntax is an easy output format for code generation. To give a flavor of DPML, the following instruction requests the same XSLT transform as in the preceding section, each "instr" corresponds with an active: URI request and each "target" is an assignment to another resource. The URI `this:response` is used as a convention to indicate the resource to be returned by the script.

```
<instr>
  <type>xslt</type>
  <operator>ffcpl:/style.xsl</operator>
  <operand>ffcpl:/data.xml</operand>
  <target>this:response</target>
</instr>
```

From this foundation it is easy to interpret the following DPML program that creates an HTML page from a database in two requests:

```

<idoc>
  <instr>
    <type>sqlQuery</type>
    <operand><sql>SELECT * from customers;</sql></operand>
    <target>var:result</target>
  </instr>
  <instr>
    <type>xslt</type>
    <operand>var:result</operand>
    <operator>ffcpl:/stylepage.xsl</operator>
    <target>this:response</target>
  </instr>
</idoc>

```

In NetKernel, language runtimes are services. Like any other service, they are stateless, and perform the execution of a program when the program code is transferred as the state. This is very different from the traditional view of software at the physical level where languages sit in front of information instead of playing a facilitating role for information. For example, to use the Groovy language runtime service, the following request provides the resource `ffcpl:/myprogram.gy` containing the program as the state for the request.

```
active:groovy+operator@ffcpl:/myprogram.gy
```

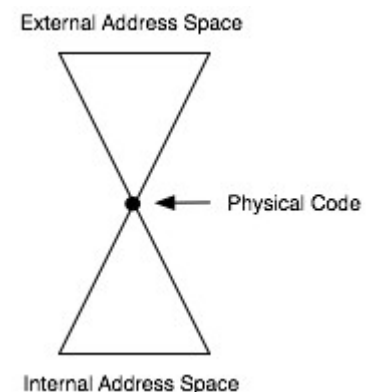
NetKernel supports a wide range of languages including BeanShell, Groovy, Ruby, JavaScript, Python, DPML XML languages such as XQuery and of course, dynamically compiled Java. Any language that runs on the Java virtual machine can be integrated into NetKernel including custom languages such as work flow engines.

Patterns

ROC presents a new set of architectural design patterns in the logical level. Let's look at two examples, Mapper and GateKeeper.

The Mapper pattern is a way to direct a bounded infinite set of resource requests to a single physical point of code. In this pattern, a request for a resource in one space is mapped to physical code which interprets and reissues each request into the mapped address space. The response of the second mapped request is returned by the mapper as the result of the first request.

This pattern has many variants, one service called `active:mapper` uses a resource containing a routing map between address spaces. Another example is the Gatekeeper which is used to provide access control for all requests entering an address space. The Gatekeeper will only



admit requests when sufficient credentials are available to validate the request.

All variants of the mapper pattern may be transparently layered over any application address space. Other uses of this pattern include auditing, logging, semantic and structural validation, and any other appropriate constraint. A particular strength of this pattern is that it can be introduced in the application without interfering with its architectural design.

Because the relationship between software in NetKernel is logically linked and dynamically resolved, interception and transformation of requests is a completely natural model. The logical address space exhibits in a very uniform way all of the characteristics found in specialist physical level technologies such as AOP.

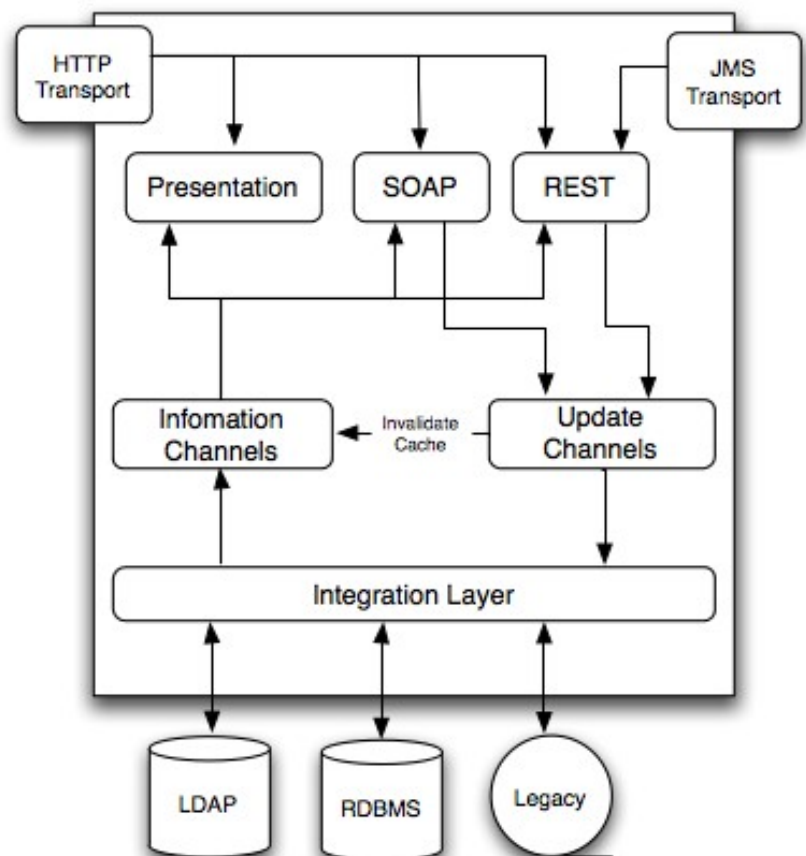
Application Development

Building applications using ROC is straight forward. If any new physical level capabilities are required, such as a new resource model, the necessary accessors, transreptors, etc. are **constructed**. Then at the logical level, applications are **composed** by identifying and aggregating resources. Finally, **constraints** are applied such as request throttles, security Gate-Keepers and data validation.

The three "C"s of ROC - construct, compose, constrain are applied in that order. This order can be reversed to make changes - constraints can be lifted revealing the composed application and allowing changes and subsequently the constraints can be reapplied. This differs from object-oriented programming where constraints are an initial consideration - classes inherently impose constraints on the use of their objects and hence the information they contain. Changes to information structure in a physical object-oriented system initiates a ripple of events - recompilation, distribution, system restarts, etc. all of which are not necessary in a NetKernel system. When compared to the flexibility of a logical system, physical level object-oriented systems appear brittle.

Application Architecture

Systems designed with physical



level technologies usually rely on mutable objects residing at various levels of an architecture. For example, object-relational mapping technologies such as Hibernate exist to create a layer of objects whose state matches that of a persistent store managed by an RDBMS. In such a design, updates are applied to objects and it is the responsibility of the mapping layer to migrate those changes to a relational database.

With ROC all representations are immutable. This leads immediately to two architectural consequences - first, caching of immutable objects can dramatically improve performance (more on this later) and second, immutable objects cannot be updated - they must be invalidated and re-requested.

While there are many valid application architectures that can be implemented with ROC, a data channels approach is commonly seen. In this design as with many, the application is composed of logically layered address spaces and passing vertically through these layers are separate read and write channels for application information. These channels might have addresses such as `ffcpl:/customers` or `ffcpl:/register-user`.

In the diagram below an integration layer translates the form of information from different sources into common structures. The read information channels support resources such as `ffcpl:/customers` which return representations of desired information. In the write channels URI addressed services such as `ffcpl:/register-user` do two things, first they update persistent storage and they invalidate any cached resource representations that depend on the update information. To developers used to the OR mapping approach (with e.g. Hibernate) this may seem very strange. In fact, it is a simple, elegant and high performance solution.

Performance

By now you must be thinking that ROC systems will spend more time running the abstraction than doing real work. However, and counter-intuitively, the ROC abstraction yields significant performance advantages.

Caching

Since every resource is identified by a URI address and the result of requesting a resource is an immutable resource representation, any computed resource can be cached using the URI address as the cache key. In addition to computed resources, NetKernel's cache stores meta information about resource dependencies and the cost of computing each cached entry. Using the dependency information the cache guarantees that cached resources are valid as long as all resources it depends upon are also valid. If a resource becomes invalid then its cached representation and all dependent resource representations are atomically invalidated.

NetKernel uses the stored computational cost information to guide it to retain the dy-

dynamic optimal set of resources - resources in the system's current working set judged valuable by frequency of use and the cost to recompute if ejected from cache. The operational result of NetKernel's cache is the systemic elimination of many redundant computations. Empirical evidence from operational systems indicates that typically between 30% and 50% of resource requests are satisfied from the cache in regular business applications. In the limit of read-mostly applications this can rise to nearly 100% giving a dynamic system with pseudo static performance. Furthermore, as the character of the system load changes over time, the cache rebalances, retaining resources that are currently most valuable.

Scaling with CPU cores

As described in the introduction, the essence of ROC is the separation of the logical information process from its physical implementation. Each request for a logical resource must ultimately be assigned to a physical thread for execution. The microkernel implementing the ROC system can optimally exploit computational hardware as it repeatedly schedules an available thread to execute each logical request. Essentially the logical information system is load balanced across available CPU cores.

Asynchronicity

ROC is innately asynchronous. The NetKernel Foundation API presents an apparently synchronous model however the microkernel actually internally schedules all requests asynchronously. A developer can therefore think with the logical clarity of sequential synchronous code while transparently gaining the ability to scale an application across wide multi-core architectures.

Additionally, accessors may be explicitly marked as being thread-safe or not, signaling to the microkernel whether it can schedule concurrent requests. This allows adoption and integration of libraries and other third party contributions without fear of unpredictable results.

Summary

NetKernel is radically different. Not, in order to create another technology stack, but in order to take a simple set of core principles (those of the Web, Unix and set theory) and extrapolate them into a coherent information processing system. In fact, NetKernel's genesis was the question, "Can the economic properties of the Web be transferred to the fine-grained nature of software systems?"

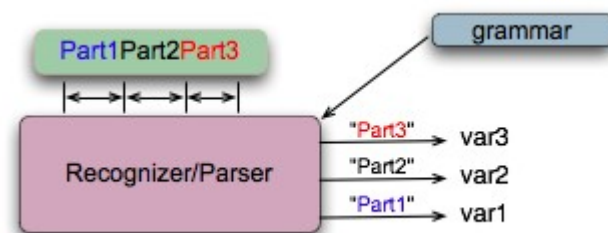
NetKernel has been production hardened over nearly eight years, from its inception at Hewlett Packard Labs through corporate enterprise architectures and even as the next generation platform for key web infrastructure (Purl.org). NetKernel has proven to be entirely general purpose and can be applied as easily to data integration, message processing or web / email Internet applications as any other application. ♠

Decoupling REST URLs from Code using NetKernel Grammars

Randy Kahle

Accessing data and services via the World Wide Web and its HTTP protocol is challenging. There have been many attempts to leverage the Web and HTTP through various designs aimed at offering efficient, concise and versionable systems - most under the umbrella of Service-Oriented Architecture. An approach that has gained a lot of attention recently, REST, relies on a URL to identify services and information. However, the Web is a dynamic, constantly changing information environment with new content and URLs being added all the time, whereas implementation code (particularly code which has been deployed and is now publicly accessible) is more difficult to change without causing problems for developers, system administrators and users. What is needed is a mechanism that can fit between the potentially fluid world of URLs and the more static world of compiled and deployed code. Such a mechanism must provide a binding between the URLs and service implementation code as well as be able to buffer and isolate the changes in the former from the code.

In software, a formal grammar is used to define the syntactic structure of textual information, such as a program, data file, or URI identifier. Programs use grammars to direct them to recognize when textual information adheres to a defined syntax as well as to parse the textual information. Pro-



grams can also use grammars to generate text that adheres to the syntactic rules. The following diagram illustrates a Recognizer/Parser program using a supplied grammar to parse the string "Part1Part2Part3" and assign the parts ("Part1", "Part2" and "Part3") to three variables.

While developing NetKernel 4 [1] we realized that a grammar based recognition and parsing technology could be used to process request identifiers and simplify software development on the platform. The NetKernel 4.0 Grammar technology [2] is a bi-directional mapping mechanism that implements this idea; it will both parse an identifier into parts and build an identifier from supplied parts. The NetKernel 4 grammar technology can be leveraged when implementing REST web services to perform the function of recognizing and

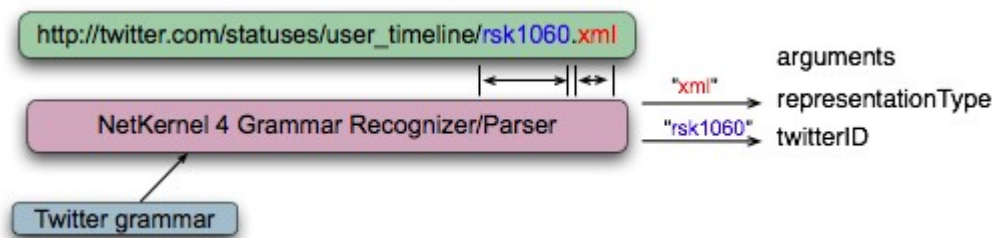
binding web service identifiers to web service implementation code.

Outside > In

To start, we will look at the use of a grammar based parser to handle the information coming from the outside, in the form of the REST web service identifier, and convert the parsed identifier text into values associated with internal named arguments. In our example we will use one of the Twitter REST web service APIs [3], which has the following general form:

```
http://www.twitter.com/statuses/user_timeline/{user-id}.{representation-type}
```

This diagram illustrates the use of a grammar driven parser to recognize the Twitter web service identifier, parse the user identification and representation information, and assign that information to the named arguments `representationType` and `twitterID`.



The following NetKernel grammar will recognize this set of identifiers:

```
<grammar>
  http://www.twitter.com/statuses/user_timeline/
  <group name="twitterID"><regex type="alphanum"/></group>
  .
  <group name="representationType"><regex>(xml | json)</regex></group>
</grammar>
```

The grammar includes fixed text ("http://..." and ".") as well as two groups. Each group defines a section of the identifier that is to be recognized using a regular expression. Because each group has a *name* attribute, the grammar engine will assign the parsed text portion of the identifier to the specified named argument. For example, the second group will recognize either a trailing "xml" or "json" and assign that value to the named argument *representationType*.

The following table illustrates how the grammar directs the parsing of example identifiers

URI	TwitterID	representationType
http://www.twitter.com/statuses/user_timeline/demo1060.xml	demo 1060	xml
http://www.twitter.com/statuses/user_timeline/pjr1060.json	pjr1060	1060

In NetKernel, an endpoint is declared with a grammar and its Java implementation class. In our example, the following endpoint declaration will cause NetKernel to associate the Twitter grammar with an instance of the Java class `org.ten60.demo.grammar.UserTimelineAccessor`.

```
<endpoint>
  <grammar>
    http://www.twitter.com/statuses/user_timeline/
    <group name="twitterID"><regex type="alphanum"/></group>
    .
    <group name="representationType"><regex>(xml | json)</regex></group>
  </grammar>
  <class>org.ten60.demo.grammar.UserTimelineAccessor<class>
</endpoint>
```

When an identifier is presented to the endpoint, the endpoint delegates to the grammar engine the job of recognizing and parsing the identifier and assigning portions of the identifier text to `twitterID` and `representationType`. Those values are available to the `UserTimelineAccessor` instance through the *context* argument of the `onSource(...)` method. The following Java code [4] is the implementation of the endpoint functioning as a reflection service [5], simply returning the information provided in the identifier:

```
package org.ten60.demo.grammar;

import org.netkernel.layer0.nkf.INKFRequestContext;
import org.netkernel.module.standard.endpoint.StandardAccessorImpl;

public class UserTimelineAccessor extends StandardAccessorImpl
{
    public void onSource(INKFRequestContext context) throws Exception
    {
        // Request the portion of the identifier that provides the Twitter ID
        String userID = context.getThisRequest().getArgumentValue("twitterID");

        // Request the portion of the identifier that provide the representation
        type
        String repType = context.getThisRequest().getArgumentValue("representa-
tionType");
```

```

        // Return a representation that simply reflects the information parsed
        from the identifier
        context.createResponseFrom("Request made for [" + userID + "] with type ["
+ repType + "]");
    }
}

```

Note that the compiled Java code is de-coupled from the structural form of the identifier. If the identifier for the service changes, a different grammar could be used to map the new identifier structure to the existing code. For example, let's say that the Twitter service introduces a version 2.0 API that provides a new way to request existing services. If the new API 2.0 URL has the form

```

http://www.twitter.com/2.0/user/timeline/status/{titter-id}.{representation-
type}

```

Then the new API can be mapped to the existing Java class with the following endpoint declaration:

```

<endpoint>
  <grammar>
    http://www.twitter.com/2.0/user/timeline/status/
    <group name="userID"><regex type="alphanum"/></group>
    .
    <group name="type"><regex>(xml | json)</regex></group>
  </grammar>
  <class>org.netkernel.UserTimelineAccessor<class>
</endpoint>

```

In NetKernel both endpoints can exist simultaneously and use the same implementation class.

Inside -> Out

Now, let's switch this around. Instead of processing requests from the outside, let's use a grammar to create requests inside our code that will allow us to access an outside service. We again use the Twitter service as our example. To create a request to the Twitter service we first define an endpoint that specifies the Twitter grammar:

```

<endpoint>
  <id>twitter:endpoint:status</id>
  <grammar>http://twitter.com/statuses/user_timeline/
    <group name="twitterID"><regex type="alphanum"/></group>

```

```

    .
    <group name="representationType"><regex>(xml|json)</regex></group>
  </grammar>
  <request>
    <identifier>res:/foo</identifier>
  </request>
</endpoint>

```

The important parts of this endpoint are the *id* and *grammar* elements (the request element must be specified but is not used in our example). The grammar element specifies the Twitter grammar that we saw earlier. The id element defines an endpoint identifier that we use in our code to retrieve the grammar. To see how this is done, look at the following code fragment from a NetKernel endpoint implementation:

```

String repType = "json";
String userID = "pjr1060";

// Create a request that retrieves and binds to the Twitter grammar
INKFRequest request = context.createRequestToEndpoint("twitter:endpoint:status");

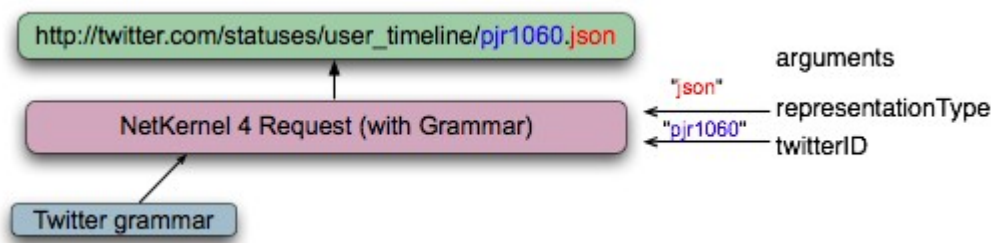
// Transfer local variable values to the named arguments in the Twitter grammar
request.addArgument("twitterID", userID);
request.addArgument("representationType", repType);

// Now we can issue a request to Twitter by issuing the constructed request
// Issue request to Twitter and capture the response
INKFResponseReadOnly response = context.issueRequestForResponse(request);

// Return the response from the external service as our response
context.createResponseFrom(response);

```

The following diagram illustrates the request object being bound to the Twitter grammar and constructing an identifier from the supplied parts.



Deep Inside

The concept of using a grammar to parse and build identifiers can be taken to the logical extreme deep within software to decouple a requestor and implementor through an associated identifier. In fact, this is exactly how NetKernel works. It borrows the idea of logical / physical decoupling from the Web and moves it inside software. Within a NetKernel system all functions are just like REST web service calls. For example, instead of making a direct API call to an XSLT processing engine, a request is made for the XSLT service using an identifier such as:

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

This URI uses the active URI scheme [6] and includes the service name, *xslt*, and two named arguments *operator* and *operand*.

Why do this? Well, the Web is malleable, but physical code is harder to change; if we introduce web-like identifiers for resources and services within our software, then our software systems can take on the properties of the Web.

Nice idea, but any reasonably experienced developer will say that the performance will be ... *\$"%^\$# ! That is a valid concern, but it misses one of the important properties of the Web - the ability to cache representations. Because real-world systems tend to follow statistical distributions, a relatively small cache of already computed values can dramatically increase overall performance. The tricky part is - for any given system, which values do you cache? This is almost impossible to predict for hand-coded memoization. NetKernel's cache [7] takes a system-wide view and balances itself as the work load changes. So, when repeated requests are made for a resource identifier, the value can be delivered from cache or computed on demand, from any available CPU core.

Companion Videos

The following video tutorials (in two parts due to YouTube's 10 minute video limit) are intended as a companion to this article, and will guide you through the NetKernel download and installation process, importing of the demonstration module defined above, and how the NetKernel Grammar debugger and Visualization tools work.

- NetKernel Grammar Part 1, <http://youtu.be/KRHHZRkQelk>
- NetKernel Grammar Part 2, <http://youtu.be/T5N1WHzprZk>

Summary

This article has introduced the NetKernel 4.0 grammar technology and shown that it provides critical flexibility at the boundary between REST web service identifiers and compiled

code. The grammar is bi-directional and can parse an identifier into named parts or build a properly formed identifier from supplied named part values. To learn more about NetKernel 4's grammar technology, download NetKernel 4 Standard Edition from the 1060 Research web site - <http://www.1060research.com>. The blog durable scope, by one of the NetKernel architects, provides insights into the design and implementation of the NetKernel platform.



References

1. The NetKernel 4 Standard Edition open platform is available for download from <http://download.netkernel.org>
2. The grammar technology is described by documentation in the NetKernel distribution and in [online documentation](#).
3. The Twitter REST API is documented on the [Twitter Developer Website](#).
4. A NetKernel module that includes the source code illustrating the use of the Grammar technology is available for download. To learn how to download NetKernel, install this module and make modifications, please view the companion video, [part 1](#), and [part 2](#).
5. The companion videos, [part 1](#) and [part 2](#), show how to augment the UserTimelineAccessor class to do more than just reflect the provided information.
6. The [active URI scheme](#) was proposed by HP.
7. A discussion about NetKernel caching can be found at [Tony Butterfield's blog](#)

A RESTful ESB implemented using NetKernel

Jeremy Deane

Background

A top-tier New England University has adopted a strategic multi-year infrastructure modernization project to replace out-of-date systems, and increase IT capabilities while maximizing the return on all IT investments. The project involves upgrading hardware, purchasing new software and training the development and operational groups. Central to the modernization strategy is the implementation of a Service Oriented Architecture (SOA).

SOA is an architectural approach to development that emphasizes an overall platform for the design of distributed applications rather than specific technologies. The mainstay of SOA is the definition and implementation of software services, regardless of location or ownership, that map directly to a system or business processes-services include interfaces and the policies that govern them at run-time. SOA benefits include loose coupling between interacting systems and platforms, a ubiquitous integration mechanism based on industry standards, support for on-demand creation of composite services and the ability to leverage existing assets while improving operational efficiency.

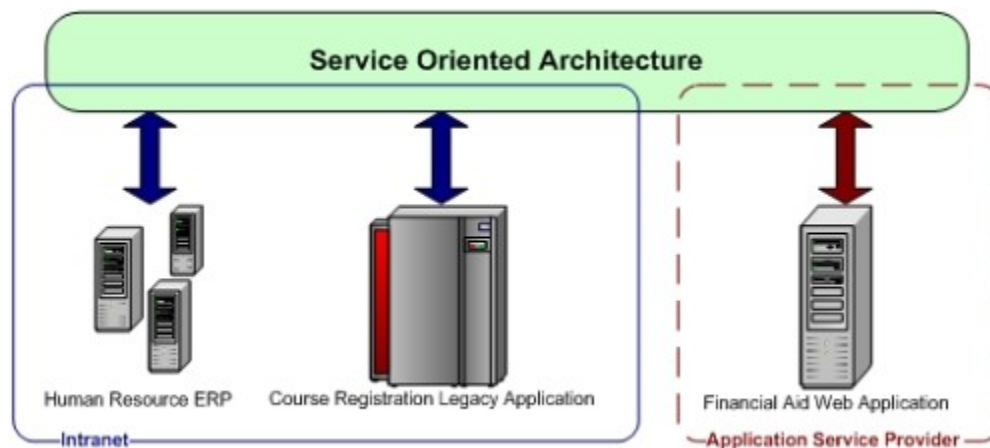


Figure 1 Service Oriented Architecture

The transition from traditional application development and deployment to a service-based approach is significant and cannot be put into practice overnight. Working with their

partner, Collaborative Consulting, the University's IT department outlined a roadmap for incremental adoption of SOA. One advantage of the incremental approach is an immediate return on investment and the ability to select the ordering of conversions to best meet the short and long terms goals of the University. The remainder of this article describes a six-month project that launched the SOA adoption process by implementing a Resource Oriented Enterprise Service Bus (ESB) using 1060 Research's NetKernel.

Problem Domain

Higher educational institutions are under constant pressure from students, staff and alumni to provide the type of on-line services they have grown accustomed to in their lives including real-time 24/7 access to information via intuitive interfaces and streamlined automated processes. At the same time there is increasing pressure from management, steering committees, and boards to control costs. Consequently, higher educational institutions IT departments such as the University's must be ingenious and pragmatic when investing in new capabilities.

The University IT department supports a wide variety of applications including Commercial off the Shelf (COTS) products such as PeopleSoft, mainframe legacy applications built on top of a Customer Information Control System (CICS) and modern J2EE web applications built using Oracle Application Server Portal. In addition, many of the applications interact with third party Application Service Providers (ASP) and provide historical information to a Data Warehouse (DW). All of these must be integrated and coordinated with the new SOA approach.

The University historically has integrated business systems using IBM MQ and FTP. This traditional approach resulted in a number of point-to-point (P2P) integrations. These P2P integrations are costly to maintain and have resulted in tight coupling between the consumer and provider. However, the existing environment was already using a light-weight message exchange state transfer (MEST) approach that is leveraged to allow for further innovation. An enterprise service bus (ESB)-a sub-type of a message bus architecture-provides for a more decoupled environment and, while it has a higher up-front cost of deployment, it was determined that the value of an ESB increases exponentially over time while the costs of extending the system remain linear and predictable¹.

The University's IT department has a small group of dedicated architects and software engineers, many of whom have developed expertise in the higher education domain through their tenure. Because the group is small, each member often takes on multiple roles including support and administration. Due to this, the IT department required a solution that would accomplish the following:

¹ [Bottom Line SOA, The Economics of Agility by Marc Rix](#)

- Meet the ever increasing consumer demand by taking advantage of reusable services and composite applications
- Reduce or eliminate the P2P integrations
- Leverage existing assets and skills for improved operational efficiency

Solution

1. Overview

Service Oriented Architectures can be implemented using a number of different patterns and technologies. The conventional approach uses patterns outlined in the WS-* specifications and one can select from a broad spectrum of technology products ranging from open source solutions such as Apache ServiceMix to commercial suites such as Cape Clear and Sonic Software. Unfortunately, the WS-* specifications are in a constant state of flux and can overwhelm developers attempting to digest over 1300 pages of detail. For example, if adhering to the all specifications, it would take the following steps/tasks to implement a SOAP service:

1. Model the process using Business Process Modeling Notation (BPMN).
2. Define a service interface using WSDL and register the service with a Universal Description, Discovery and Integration (UDDI) repository.
3. Generate Business Process Execution Language (BPEL) scripts using the BPMN that access the services from the service registry.
4. Define policies governing access to the service using WS-Policy.

The commercial ESB suites in the market were evaluated and since the IT group is relatively small, the decision was made to look for a solution that would result in a low-friction system that could foster the type of innovation a small group of IT resources could undertake and wouldn't force the group into a centralized service ownership model that relied on a single vendor. The University's domain is extremely fluid, with process changes, application changes and integration changes; therefore what was needed was an overall architecture and strategy that was a reflection of the true nature of the university.

Since message requests already were established as the central transfer mechanism across the university, a RESTful or Resource-Oriented approach was employed to implement an SOA. REST is based on a small set of widely-accepted standards, such as HTTP and XML and requires far fewer development steps, toolkits and execution engines. The three key benefits of a RESTful approach to SOA include a lower cost of entry, quicker time to market, and flexible architecture. A Resource-Oriented approach goes beyond a RESTful approach and provides a deeper more extensible and transport independent foundation. While REST design patterns advocate the use of HTTP a Resource-Oriented architecture supports services connected to HTTP as well as transports such as JMS or SMTP.

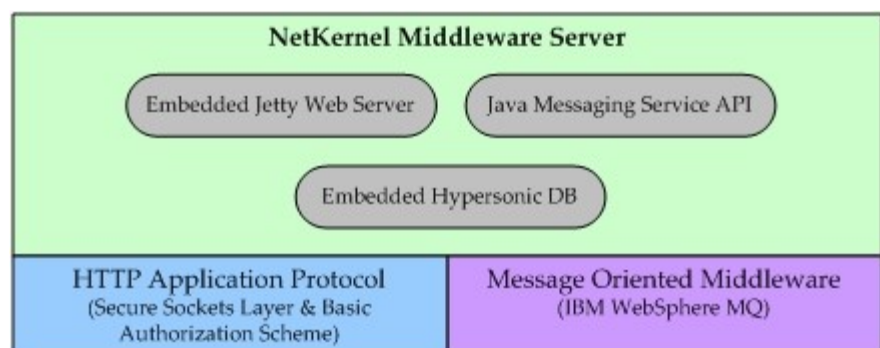
Although several ESB implementations such as Codehaus Mule support REST, only 1060 NetKernel is built upon a Resource-Oriented Computing Platform (hence, “ROC”). The core of Resource-Oriented is the separation of logical requests for information (resources) from the physical mechanism (code) which delivers the requests. Services built using ROC have proven to be small, simple, flexible and require less code compared conventional approaches. This made it the ideal technology to build a technology platform.

2. Technology Platform

NetKernel is resource-oriented middleware that provides core Enterprise Service Bus (ESB) capabilities, addressing, routing and data transformations and can act as a service registry orchestration engine. NetKernel is a rich offering and also provides advanced capabilities for XML transformations, cache management and multi-threaded processing along with multiple transport protocols, including HTTP and JMS and a SOAP engine enabling it to provision conventional web services. NetKernel is a solid foundation for heterogeneous enterprise integration.

Conceptually, NetKernel provides access to resources identified by a Universal Resource Identifier (URI) address. All URI addresses are managed within a logical address space. A REST-based micro-kernel handles all requests for resources, resolving the URI from the address space and returning a representation of that resource. Requests to the micro-kernel can also be made to create new resources or update or delete existing resources.

Physically, a NetKernel system is composed of modules which expose public service interfaces and resources via URI addresses. Similar to a Java EAR, a module contains source code and resource configurations. Modules may logically import another module's publicly exposed services and resources incorporating them into the address space. Since imports refer to a module's logical name and can specify a version number, multiple versions simultaneously can be run and updated in a live system. Service requests are injected into NetKernel by transports, which are event detectors residing at the edge of a system. Service consumers can send requests via any supported transport such as HTTP or JMS.



HTTP is a stateless request-response application protocol. The request message structure is comprised of a command, headers, and a body. The response message is comprised of a status, header and a body. Clients and servers exchange these messages using Transmis-

sion Control Protocol (TCP) sockets. The client-server interactions can be secured at the transport layer using the Secure Sockets Layer (SSL) protocol while the message itself can be secured using encryption and a digital signature. Finally, a client can be authenticated using HTTP-Basic or HTTP-Digest authentication schemas².

The Java Message Service (JMS) API provides the platform for implementing asynchronous services. However, the API requires a provider, in this case IBM WebSphere MQ. IBM MQ is Message Oriented Middleware (MOM) and provides queue-based communication channels among applications. Channels are implemented using Point-to-Point or Hub & Spoke topology. In addition to transporting messages, MQ can also handle workflow, process automation, data transformation, monitoring and management.

3. Resource-Oriented Services

A Resource-Oriented service provides transport-independent stateless access to a resource. A resource is an abstraction of information. For example, a student's registration is an abstraction which can have representational forms such as a web page, an XML document, or a PDF file. A service exposes each resource representation using a resource identifier or address. The resource identifier is actually a relative Universal Resource Indicator (URI). A URI is composed of two parts, the scheme (e.g. HTTP and FTP) and the address. The second part of the URI is relative. The address, /domain/account/45322, is relative until it is associated to a scheme such as http, <http://domain/account45322>.

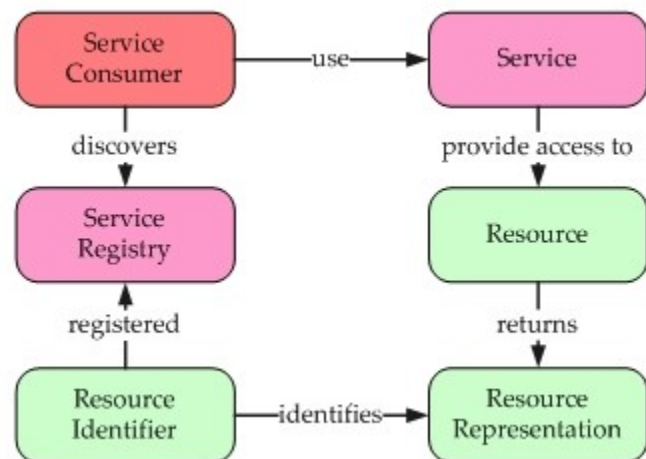


Figure: Resource Oriented Service Conceptual Model

The service defines a set of actions, Create, Read, Update, and Delete, that can be invoked on a resource. In addition, certain actions may trigger rules or business logic. When a resource is requested, a physical immutable representation of that abstract resource is returned. Different types of representations can be returned based on business logic, such as the type of consumer. For instance, most back-end processes require an XML document while a front-end process may require a JSON object. The service also can receive a representation creating a new resource, or updating an existing resource. Finally, the service can receive a request to delete a resource.

² [HTTP Authentication: Basic and Digest Access Authentication](#)

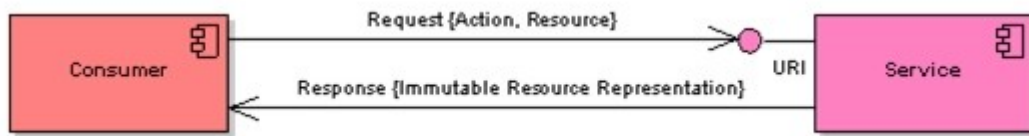


Figure: Resource Oriented Service Interaction Model

A transport resides at the edge of a system and when it detects an event it issues a corresponding internal root request. For example, the HTTP transport listens for a consumer's request for a Uniform Resource Locator (URL), specifying an access method (e.g. GET, PUT, POST, & DELETE). The URL is transformed into an internal relative URI and the access method is translated into an action. In the case of a request over JMS, the URI and action are passed as message header parameters. In addition, if a resource representation should be returned, a return queue parameter is specified in the header.

In a RESTful system, access to resources is stateless which means each request must have a way to pass context as meta-information. In general, the transport defines a message structure comprising a header and a body. The header is used to pass the meta-information while the body is used to pass the resource representation. For instance, if a Resource-Oriented service is exposed over HTTP, authentication information can be passed in the header, and if the service is exposed over JMS that same information can be passed as an encrypted header parameter.

Resource-Oriented services are built using Maven 4 and packaged as NetKernel Modules. Maven is a tool for building and managing software within the context of a “project.” Maven projects are defined by a Project Object Model (POM) XML. The POM defines the software's dependencies, build process and deployment structure (e.g. JAR, WAR, EAR). In addition, a Maven project can be used to generate a project web site and deploy software. In this case, Maven packages services within a NetKernel Module and publishes information about those services to a registry.

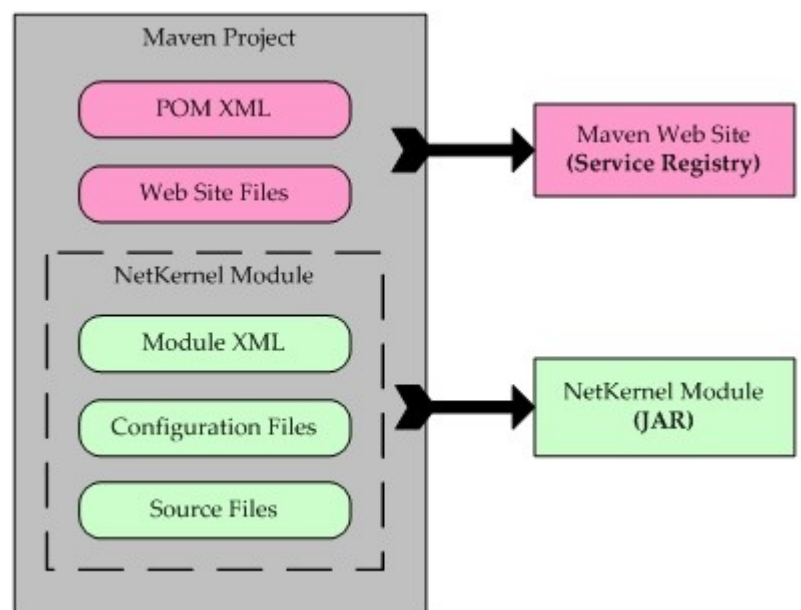


Figure: Resource Oriented Service Development

4. *Resource-Oriented Enterprise Service Bus (ESB)*

A Resource-Oriented Enterprise Service Bus (ESB) was implemented using NetKernel. At the heart of NetKernel is a RESTful or Resource-Oriented microkernel responsible for resolving logical URI requests to physical code endpoints and scheduling the request on an available CPU core. While the mapping of logical address to physical code is defined in the application structure, the actual binding of logical address to physical code occurs only for the duration of a request and is then discarded.

Because each request to the micro-kernel is bound anew, system administrators are free to change the association between logical address and physical code while the system is running enabling capabilities such as real-time live code updates. Performance in practice is paradoxically not degraded by this indirection and binding but rather enhanced as the URI address acts as a key to NetKernel internal cache. If any resource is re-requested and dependencies have not changed then the cached representation is returned instead of being re-computed.

The use of a Resource-Oriented microkernel has several key benefits. First, service interaction is at a logical rather than physical level. This results in loosely coupled interactions, thereby decreasing the impact to the consumer and provider when changes are made to the physical implementations. Second, the results of requests are cached, thus decreasing the overall cost of service composition and orchestration. For instance, if a set of orchestrated services relies on a common service, the underlying physical code of that common service seldom will be executed. Finally, all internal requests to the microkernel are asynchronous, allowing processing to scale linearly as more CPUs are added to the host server.

The ESB is primarily responsible for service provisioning and security. Service provisioning involves exposing Resource-Oriented services to consumers over transports such as HTTP and JMS. The transports map the external URI and access method to an internal Resource-Oriented service and action. Regardless of the transport, the body of the request, such as an XML Document or JSON object, is passed as a parameter named 'param.' Consequently, the Resource-Oriented services are decoupled from details of transport specific logic and in fact, additional protocols can be added at any time without impacting existing code.

∕

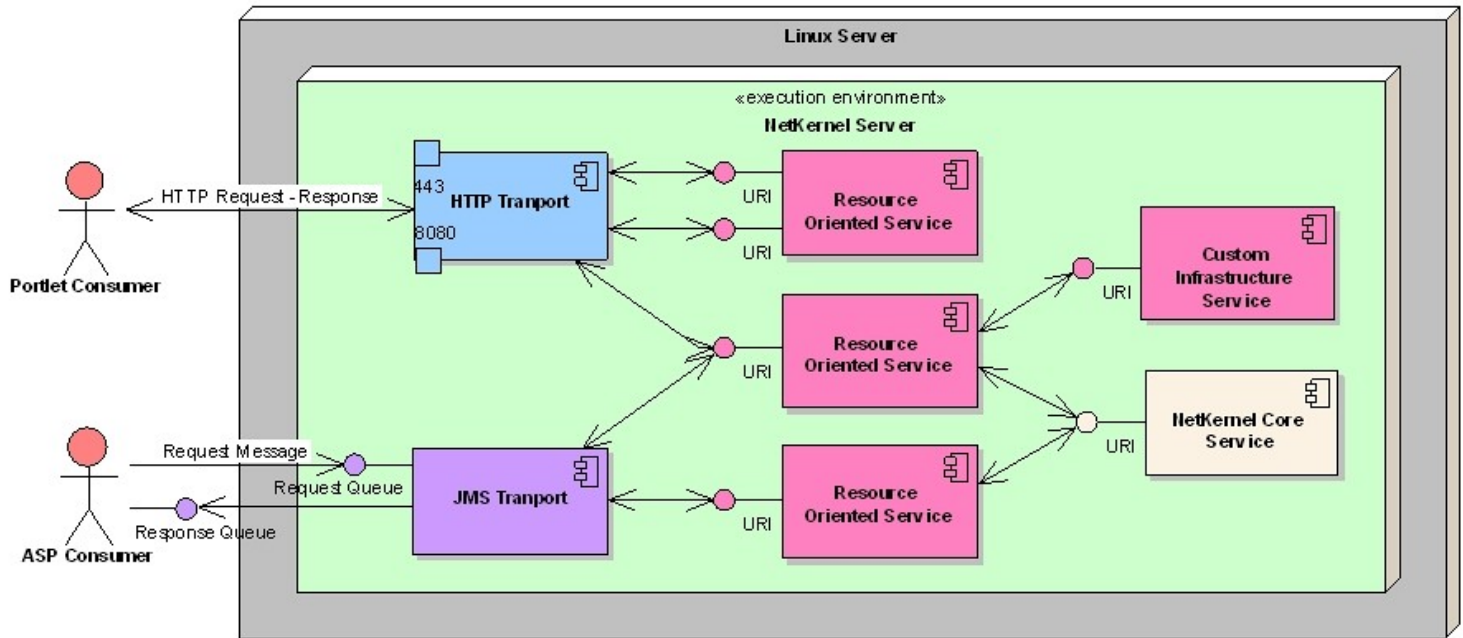


Figure: Resource Oriented Service Provisioning

The Resource-Oriented services, in turn, delegate to a set of custom infrastructure services and core services provided by NetKernel. NetKernel provides an extensive set of core services. For instance, core services exist for XML and SOAP processing, CRON Job scheduling, and SMTP interactions. The core services dramatically reduce the amount of code that needs to be written to implement a Resource-Oriented service. Custom infrastructure services are used to expose capabilities which can be leveraged in the higher education domain.

Each request to the ESB is first authenticated, then authorized and (in some cases) audited. Transports authenticate an incoming request based on a user-name password combination and then delegate authorization and auditing to a security service. Authorization involves verifying that the identified consumer has the appropriate privilege. A privilege is comprised of a relative URI and an action. As an example, a consumer could be authorized to read but not to update or delete the resource student profile, identified by the relative URI /domain/student/identifier/profile. Unauthorized requests automatically are audited and there exists the option to audit based on the identified consumer or privilege. The account, privilege and audit information is stored in an embedded Hypersonic database.

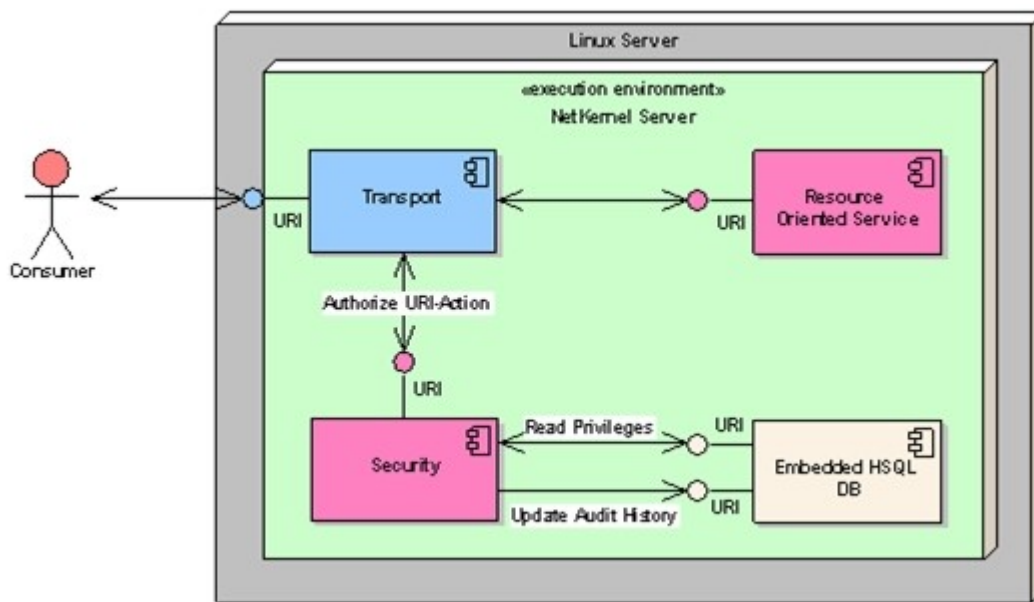


Figure: Resource Oriented Service Security

Summary

Other than a run-time service governance solution, such as Actional or ManagedMethods, middleware for implementing an ESB is essential to any successful SOA initiative. Without an ESB, an organization is simply increasing the number of costly point-to-point (P2P) interactions using web services. While there is wide disagreement as to the composition and purpose of an ESB, most can agree on a set of core capabilities including service addressing, message transformations and message routing. An ESB implemented using NetKernel middleware provides these capabilities and more advanced capabilities such as service registration and orchestration.

The NetKernel product allowed the University to implement a Resource-Oriented ESB. A Resource-Oriented ESB is essentially an open standards-based enterprise integration framework. This framework enables an enterprise to reduce or eliminate costly point-to-point interactions and decreases the time to market for introducing new capabilities. Furthermore, this framework has a lower initial cost of entry than a conventional enterprise integration framework based on the WS-* standards. Additionally, since NetKernel and ROC provide for integration on a per-service basis, the University can push the integration functionality to the edge of the network (as a URI), which translates into better service management and scalability. In short, this framework provides an organization unprecedented enterprise architectural agility.

In less than six months, a team of three software architects was able to implement a Resource-Oriented ESB and several initial Resource-Oriented services using NetKernel middleware. The successful implementation of the ESB launched the University's incremental

adoption of SOA. A Resource-Oriented approach allowed the team to leveraged existing assets and technical skills. Going forward, the University's IT department now has the ability to meet the ever increasing consumer demand by taking advantage of reusable services and composite applications while reducing or eliminating the P2P integrations.

About the Author

Jeremy Deane is a Technical Architect at Collaborative Consulting. He has over 12 years of software engineering experience in leadership positions. His areas of expertise include Enterprise Architecture, Performance Engineering and Software Process Improvement. In addition, he has Masters in Information Systems from Drexel University and has recently published a white paper, Document Centric SOA. ♠

Resource-Oriented Analysis and Design: Parts 1 – 9



Resource Oriented Analysis and Design

Part 1

Peter Rodgers, PhD

Background

People who have learned ROC will tell you "ROC is not difficult, just different". But over the last year I've seen a number of cases where developers immersed in state-of-the-art best practices for "coding" have spectacularly bounced off. Whilst equally, in the same time I've seen people with no experience of ROC rapidly develop and deploy to production massively scaled heavily loaded mission critical systems that "just work".

The latter cases were achieved by going with the flow, relaxing and letting the system take the strain and, fundamentally embracing "Composition" (the second of the 3C's). The bounces seem to happen when the initial starting assumption is "Construct" - ie "we build software by writing code".

The challenge for anyone embracing ROC is that this latter proposition is actually the current industry-best-practice. For example, we see it in the practices of Test Driven Development (TDD)... before you do anything write (code) a test which fails then write (code) some code that makes the test-code pass. repeat... code test-code, code code, code test-code, code code...

One of our challenges in the next phase of tooling in NetKernel is to introduce a set of technical measures that will attempt to meet half-way the working practices of "regular coders". In short to make things feel more like they're used to them feeling. Whether its IDE integration, testing frameworks, classpaths etc. (See below for an example of concrete steps that are underway).

But there is a fundamental dilemma - since to make ROC rigidly fit into the orthodox tooling would be to constrain its power and flexibility. It is precisely because "Constraint" (the third C) is last in the three C's that ROC solutions are malleable, adaptive, fast to create, easy to evolve. With ROC anyone who's ever learned it will tell you "change is cheap". (the fourth and fifth C's!).

Anyway - that is a little context for what follows, and it has resulted in me reading around and examining the "classical paradigm".

Recently, I came across an often cited example called "TDD as if you meant it"... <http://gojko.net/2009/08/02/tdd-as-if-you-meant-it-revisited/>

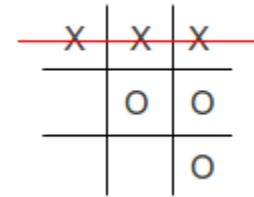
An article about a workshop teaching the TDD methodology. I realised that the domain problem which is chosen, to implement "Tic Tac Toe" ("Noughts and Crosses", to Europeans), would equally serve as an exercise in showing how to think and work in ROC. Also, since the teams in the article have provided the source code of their solutions, maybe we can get a sense of comparison.

What follows is, as near as I can make it, a stream of consciousness description of my approach to the TicTacToe problem. Along the way I hope to show the balance between C1 and C2 (construct and compose) and how C3 (constraint) is used sparingly and late in the process. In doing so I also hope to show, what I've recently come to think of as, ROC's "Relevant Testing".

Who knows, maybe I'll pull it off, maybe I'll crash and burn? As I write I'm winging it and have written absolutely no code. All I've done is spend half an hour thinking about the problem and the resources. So lets get started...

Tic Tac Toe Resource Model

The game of Tic Tac Toe (Noughts and Crosses) is one you no doubt learned when you were a young kid. A two-dimensional array provides a grid of 9-cells in which you may mark your token (a nought O or a cross X). Each player takes a turn to mark a cell. The winner is the first to get three consecutive tokens in a straight line.



A mathematician would be able to tell you that there are a couple of hundred thousand possible games. The first player to go has a two-to-one statistical advantage! And, rather sad for your inner child, achieving a draw is assured unless your opponent makes a mistake.

But you know the rules, that's the whole point. We don't need to spend any time defining the domain problem.

Classical Approach

When you read the TDD article you will see that they also waste no time on the model - but instead get to the definition of the constraints. Which we'll quote directly (I changed their term "field" to cell since I just think its the more precise technical name of a lattice grid)...

- a game is over when all cells are taken
- a game is over when all cells in a column are taken by a player
- a game is over when all cells in a row are taken by a player
- a game is over when all cells in a diagonal are taken by a player
- a player can take a cell if not already taken
- players take turns taking cells until the game is over

This is a Constraints-first approach to defining the problem. It is unstated and implicit that there is state associated with a cell and indeed the whole playing grid. There are also a whole set of implicit and presumptive terms (row, column, diagonal) - of course we know what these mean since we know the problem intimately (but don't forget the implicit presumptions going on at this point).

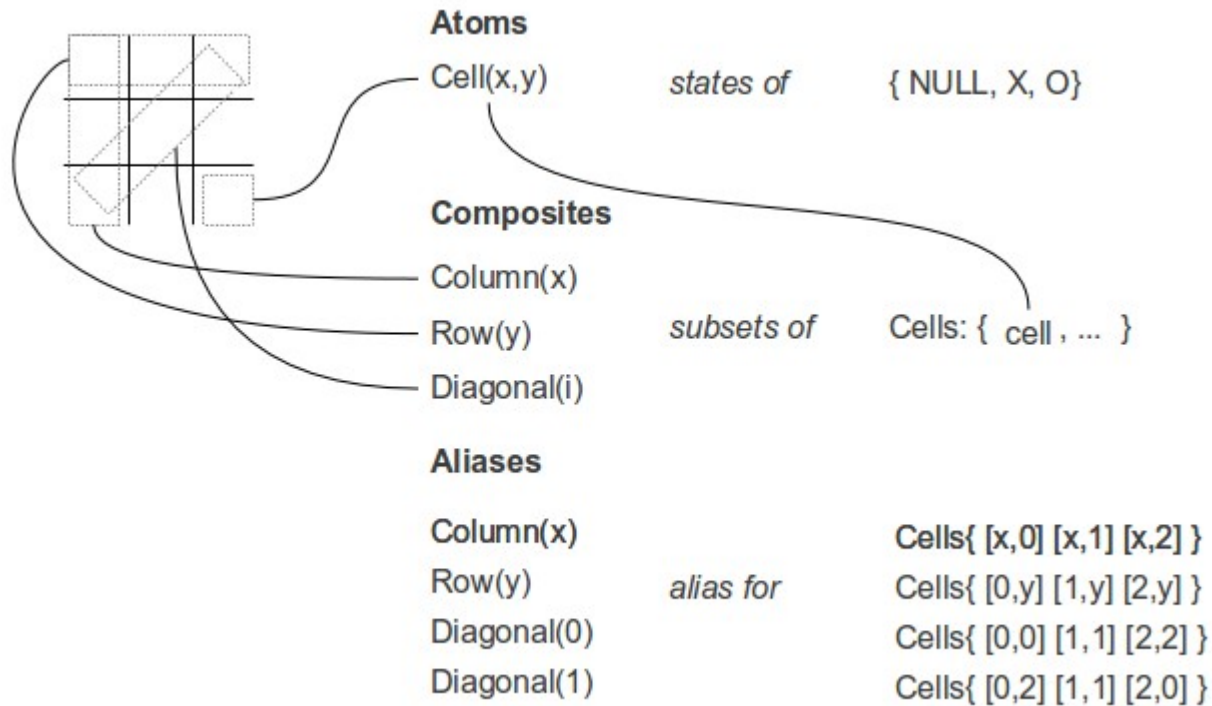
No doubt those trained in Object Oriented design are already envisaging a Cell class with associated getters and setters, a Game class containing a structured set of Cell members and an interface of methods which could be called to determine if the constraints above are applied etc etc. Lets call this the "classical approach" - we'll undoubtedly come back to this later in the series of articles.

So now lets switch to the ROC perspective. At first it might look quite similar, but after only a very short transition we start to look at things in quite a different way...

ROC Approach

First we start by thinking about the resources and their possible state. What we're aiming to find are the Sets and the memberships of sets and any associated compositional Sets (composites).

Here is a diagram and definitions of the resources of the problem (*this is a tidied up copy of something I sketched up on the whiteboard in ten minutes*)...



Resource

The first thing we notice is that a Cell resource is the "atom of state". We see that it may have states of "NULL, O and X". So far so very like OO-modelling.

But the very next thing to notice is that every Cell has an identity. In fact I started by thinking - "what is the name of a cell resource". I have not worried about an ROC URI name (we can choose our naming convention later) but what I am saying is that a cell has an identity **Cell(x,y)** and x and y are grid reference co-ordinates.

So lets define Cell(0,0) as the identity of the resource "top left cell" and Cell(2,2) is the identity of the resource "bottom right cell".

Composite Resources

We know from our knowledge of the domain problem that the outcome of the game depends upon the concepts of "row", "column" and "diagonal" - indeed to win requires that the state of one these should be all X's or all O's. Immediately we have identified three new sets of resources and so we can give them identities. Row(y), Column(x) and Diagonal(i).

But with a little more thought we realise that Rows, Columns and Diagonals are not atomic resources. They are composite resources consisting of small sets of atomic Cell(x,y)

resources. In short we see that they are subsets of the set of all Cells{ }.

Immediately we realise that we now need to define a new composite resource Cells{ ... } comprising a set of Cell(x,y) resources and which therefore allows us to give an identity to any subset of cells (hang with it - it will become very clear in a second).

So when we refer to Row(0) what we're really doing is talking about the identity of a set of cells. Similarly when we talk about Column(1) this is another set of cells. Row, Column and Diagonal are just aliases to specific subsets of Cells{...}.

It follows that we don't need to "write any code" for Row, Column or Diagonal - we can just declare a mapping from the aliased identity to the explicit identity. So...

Column(x) is an alias for Cells{ Cell(x,0), Cell(x,1), Cell(x,2) }

which, to be explicit means that ...

Column(1) is an alias for Cells{ Cell(1,0), Cell(1,1), Cell(1,2) }

similarly, you can see in the diagram we can define alias mappings for Row(y) and the Diagonals.

Pause a moment and think about this.

- We have defined one set of atomic resources: Cell(x,y)
- We have defined one set of composite resources: Cells{...} (which references atomic Cell(x,y) resources).
- We have defined three sets of composite resources that have "meaning to our game" but which are really just aliases for subsets of the Cells{...} resource set.

Maybe we might want to start to think about new games in which the concept of "The Corners" has meaning - no problem its just another named subset of Cells{...} but lets not get ahead of ourselves, we've not written TicTacToe yet...

What we're doing here is saying "giving things identity is powerful" it allows us to refer to things collectively - that is, it allows us to deal with whole sets rather than small parts. (Remember when in a previous newsletter I said "the bag is man's greatest invention", for this same reason? We just rediscovered this).

OK now we've conceived the fundamental resource model we can build on it to define some more meta resources that have relevance to the game of TicTacToe...

Meta Resources

WinningSet	Cells{ [] [] }
History	Cells{ ... }
FreeSpace	9 - Size of History
GameOver	!(WinningSet == {}) or FreeSpace==0
CheckSet(x,y)	Column(x) Row(y) if(OnDiagonal(x,y,0)) Diagonal(0) if(OnDiagonal(x,y,1)) Diagonal(1)

We know that its important to know if the game has been won. By sticking to the mind-set of "Sets of Cells" we realise that there is a meta resource which we can call the WinningSet - it is a composite resource and is another subset of Cells{...}.

At the start of the game the WinningSet is empty. When the game proceeds, eventually we will want to hold the state of the winning row, column or diagonal - the WinningSet will be made real. But we don't need to bother ourselves with writing any code to do this yet - its sufficient for the moment that we have the concept of "WinningSet" and understand it as an unreified subset of Cells{...}.

Equally we can see that having a history of played moves is a useful resource. Again it is a meta resource and is just a list of Cells (another subset of Cells{...}).

So now we can define the **GameOver** resource - this is true if and only if the WinningSet is not empty or there is no FreeSpace left to play. We already understand what the WinningSet is and its easy to see that the FreeSpace resource is just 9 minus the size of the History set.

Finally we have a resource which is really alien to the classical domain. When we play a move we need to decide if it was the winning move. So we need to check to see if it fills a row or column or diagonal. We can define the CheckSet(x,y) for a Cell(x,y) as being the identities of the sets we need to go and check. So the resource CheckSet is a "linked data" resource!

Say we play Cell(o,o) then CheckSet(o,o) would be the resource:

{ Column(o), Row(o), Diagonal(o) }

The **set of identities** of the set of resources we need to know the state of to see if we've

reified the WinningSet.

This last concept of a set of linked data resources is not so far from the Cells{...} (also linked data resource!) but you might be beginning to feel a bit light headed!

Next time we'll start to implement the solution and it will crystallize. But you may already be beginning to sense that without writing any code we're on the path to a normalized solution - the logic we will need to execute to play the game will be the bare minimum. (You might want to peak ahead and compare and contrast where we're leading with the "repeated iterations of cells" designs seen in the OO of the classical solutions in the TDD exercise). ♠

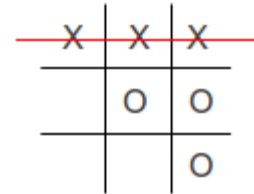
Resource Oriented Analysis and Design

Part 2

Peter Rodgers, PhD

Recap

In part 1 we started to consider a Resource Oriented model of the TicTacToe problem. Here's a quick recap...



- We identified that a cell on the board could be considered as the primary atomic resource.
- We then discovered that the higher level resources could be considered as sets of cells - which we called "composite resources".
- Finally, with a low-level resource model shaping up, we set our mind on the higher level resources that would be necessary to implement the TicTacToe game. Again we found that these were also composite resources - we also started to see that the representation state of these resources could simply consist of sets of identifiers to other composite resources. We called the resources associated with the TicTacToe game, "meta resources" and saw that from an ROC perspective they are metadata.

Up to now we've used high-level function-like notation as the resource identifiers - as a sort of "casual short-hand" while we got our heads around the resource model. Today we're going to examine how, over a thirty minute period, I went about turning the conceptual model into URI addressable resources...

TicTacToe in ROC: Defining the resources

My first step was to create a new module which, using our URN-style reverse domain name convention for naming, I called `urn:org:netkernel:demo:tictactoe`. I could have created a module with the new module wizard - but its as easy to create a completely empty module with the bare minimum content by cutting and pasting another `module.xml` from any old module I had to hand. I blew away its rootspace with an empty one and changed its meta-data to get this...

```
<module version="2.0">
  <meta>
    <identity>
      <uri>urn:org:netkernel:demo:tictactoe</uri>
```

```

    <version>1.1.1</version>
  </identity>
  <info>
    <name>Demo / Tic Tac Toe</name>
    <description>Tic Tac Toe in ROC</description>
  </info>
</meta>
<system>
  <dynamic />
</system>
<rootspace />
</module>

```

In the physical-sense my starting point is a completely empty space as it defines no end-points. Alternatively, with an ROC perspective and remembering that in ROC we think of resources as abstract, we could look at it entirely the opposite way - this space contains potentially all possible abstract resources. Our job, just like that of a sculptor, is to "make the right moves" so that the possible set of resources is constrained to ensure only those we care about in our resource model can be reified.

Excuse the philosophical mind-games but this is relevant. What I'm trying to highlight is that if you are a "coder" - you will think of the job of solving the problem as one of "adding physical code" until we get the information we want. Whereas if you can switch to thinking in terms of resources - you can imagine that the information you want is already in there and your job is to release it by "trimming away" what we don't want to leave us what we do. If you can learn to switch between both perspectives you open yourself up to many powerful opportunities...

With my new module constructed I registered it with NetKernel by adding a reference to it in a file called [**install**]/etc/modules.d/tictactoe.xml. I really love the new modules.d/ capability and use it for grouping all my applications into different collections that I can load and unload. For example changing the file extension .xml -> .xml.hide will unload a set.

My next step was to immediately copy my new module (cut and paste the directory) to create a complementary test module. I renamed the copy and give it the name urn:test:org:netkernel:demo:tictactoe - the same name as the demo but with a "test" prefix. After changing the identity and descriptive metadata in its module.xml I also registered this with NK by cutting and pasting the line in tictactoe.xml for the first module and adding the "test" prefix. I then checked with the space explorer that my two modules (and two empty address spaces) were there and NK was happily managing them for me. For the record, here's my tictactoe.xml file...

```

<modules>
  <module runlevel="7">../urn.org.netkernel.demo.tictactoe/</module>
  <module runlevel="7">../urn.test.org.netkernel.demo.tictactoe/</module>
</modules>

```

Total elapsed time: Somewhere less than one minute.

Setting up the XUnit Test Context

You can skip this section if you wish - its detail which you may already know about, or if you use the New Module Wizard has been automated for you. But if you want to know exactly what I did, then this is the literal story...

I now set-aside the first module and put all my focus on the test module. I created a directory structure with paths /etc/system/ and /test/. I copied and pasted a Tests.xml file from an existing test module into the /etc/system/ directory. I edited Tests.xml file to look like this...

```

<tests>
  <test>
    <id>urn:test:org:netkernel:demo:tictactoe</id>
    <name>TicTacToe Tests</name>
    <desc>Tests for the TicTacToe Demo</desc>
    <uri>res:/test/testlist.xml</uri>
  </test>
</tests>

```

This is the metadata description for a set of XUnit tests. Now I needed to make sure that my rootspace exposed this resource so that the test framework would find this test set, so I added a <fileset> pointing to Tests.xml, to the empty <rootspace>...

```

<rootspace>
  <fileset>
    <regex>res:/etc/system/Tests.xml</regex>
  </fileset>
</rootspace>

```

You can see that the test set in Tests.xml has a link to the <uri> of the actual test declaration resource that will be used when this set of tests is actually run. Usually I start off calling my test list the same name every time res:/test/testlist.xml and I know that I've got to add this, so I created a testlist.xml file to the test/ directory which looked like this...

```
<testlist />
```

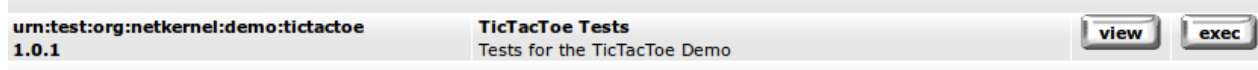
And then made sure this would be resolvable by exposing the files under the test/ directory to the ROC domain with another <fileset> in the rootspace...

```
<rootspace>
  <fileset>
    <regex>res:/etc/system/Tests.xml</regex>
  </fileset>
  <fileset>
    <regex>res:/test/.*</regex>
  </fileset>
</rootspace>
```

Now I took a look at XUnit to make sure my test set was being automatically discovered and aggregated into the test framework

<http://localhost:1060/test/>

which shows this test set...



Finally, I have one thing left to do. I need to import my real module so that its resources can be requested by my tests. Which leaves me with this as my contextual test space...

```
<rootspace>
  <fileset>
    <regex>res:/etc/system/Tests.xml</regex>
  </fileset>
  <fileset>
    <regex>res:/test/.*</regex>
  </fileset>
  <import>
    <uri>urn:org:netkernel:demo:tictactoe</uri>
  </import>
</rootspace>
```

I have tried to show the precise steps I took. Including how I went to the tools to verify each of my changes. If you like, the New Module Wizard will do all these steps for you in one go - for your convenience it puts the test space into the new module too. However, our con -

vention is to keep tests separate so we always have a separate test module for each of our production modules.

Total elapsed time: Somewhere less than two minutes.

The Art of Thinking of Good Names

The real work starts here. I now had to think of how I would turn the casual identities I'd described in part 1 into useful, long-lived and stable names in my ROC solution.

The first thing I did was procrastinate. I deferred the work and simply created a test with literally the same casual name I'd used already. I added a test requesting **cell(o,o)**...

```
<test name="cell(0,0)">
  <request>
    <identifier>cell(0,0)</identifier>
  </request>
  <assert>
    <exception />
  </assert>
</test>
```

This test asserts `<exception>` since, as we know, the tictactoe `<rootspace>` is still empty and so this request will not be resolvable. I tabbed over to the XUnit view, ran the tests and received a satisfying 1 test run 0 failures report.

I like to always keep things "green" and so asserting that I know this will fail allows me to know that I am on top of things - there are no unknowns - I expected this. If you prefer to have your tests fail you could assert `<notNull>` which would then lead to a failing "red" test.

While what I'd proposed so far was a perfectly legitimate identifier and would serve as a name for a cell resource, I was uneasy and knew that it was only a temporary step. But I still procrastinated...

Next I considered the `cells{...}` composite resource from our model. Again I avoided the work of thinking of a good name and instead just created a second test like this...

```
<test name="cells{cell(0,0),cell(0,1),cell(0,2)}">
  <request>
    <identifier>cells{cell(0,0),cell(0,1),cell(0,2)}</identifier>
  </request>
  <assert>
```



```

    <exception />
  </assert>
</test>

```

My initial unease was manifest. This was not a nice looking thing at all. All those commas and mixtures of brackets!

As I had already intuited, I decided that what was needed, since we were going to have composite resources that would themselves be linked data to cell resources, was a really compact identifier for the cell resources.

I played around with `res:/cell/x/y` and `cell:/x/y` but these were still too verbose, so I settled on `c:x:y` - which I think is probably the most compact name you could have while preserving the implicit structure.

Through a bit of search-replace my tests became...

```

<testlist>
  <test name="c:0:0">
    <request>
      <identifier>c:0:0</identifier>
    </request>
    <assert>
      <exception />
    </assert>
  </test>
  <test name="cells{c:0:0,c:0:1}">
    <request>
      <identifier>cells{c:0:0,c:0:1}</identifier>
    </request>
    <assert>
      <exception />
    </assert>
  </test>
</testlist>

```

Progress, and perfectly legitimate identifiers. NetKernel doesn't care what you call anything. All identifiers are just opaque tokens to the kernel...

...this is true. But even though its convenient for shorthand, just as when mapping REST paths to internal service implementations, I like to know that there is a normalized and stan-

standard form of active: identifier for implementing services. So I added two more tests like this...

```
<testlist>

    ...existing initial c:0:0 and cells{} tests...

    <test name="active:cell+x@0+y@0">
        <request>
            <identifier>active:cell</identifier>
            <argument name="x">0</argument>
            <argument name="y">0</argument>
        </request>
        <assert>
            <exception />
        </assert>
    </test>
    <test name="active:cells+operand@c:0:0,c:0:1">
        <request>
            <identifier>active:cells</identifier>
            <argument name="operand">c:0:0,c:0:1</argument>
        </request>
        <assert>
            <exception />
        </assert>
    </test>
</testlist>
```

By doing this I had determined that I would have the primary implementations as active:cell and active:cells (plural) each with appropriate and normalized arguments. I didn't waste time trying to think what the argument for a collection of cells ought to be called and went with the convention of just using "operand".

You'll note that I am expecting the argument to be an identifier reference consisting of a list of cell identities. This is because in our part-1 analysis we'd recognised that certain sets of cells would be aliased to shorthand names (row, column, diagonal). By using a reference identifier as the argument it leaves me free to create these aliases with no code. I can "get away with this" since the possible range of identities is pretty small in this problem - in a larger resource space I would probably have gone with the operand as a resource transreptable to HDS containing a list of cell identities. We might see later that we may also, for completeness, require a service with an explicit pass-by-value argument (let's see).

At this point I had a set of tests and the requests they declared defined the resources my tictactoe space needed to reify. Time to switch over to the demo module...

Total elapsed time: Somewhere around ten minutes.

Implementing the Resources

Before I switched my attention back to the main tictactoe space, I quickly changed all my asserts in my tests to `<notNull>` - since now my attention had changed from defining identifiers to actually having my test requests resolve to tangible resources. So now I would know when I had my space configured with correctly resolvable resources.

With this change, my tests were now all red.

In the tictactoe space, my first step was to add a mapper like this...

```
<mapper>
  <config />
  <space />
</mapper>
```

I then stubbed out the first normalized active:cell service which consisted of an active grammar definition and a dummy request...

```
<mapper>
  <config>
    <endpoint>
      <grammar>
        <active>
          <identifier>active:cell</identifier>
          <argument name="x" />
          <argument name="y" />
        </active>
      </grammar>
      <request>
        <identifier>res:/dummy</identifier>
      </request>
    </endpoint>
  </config>
  <space />
</mapper>
```

By way of scaffolding I simply mapped `active:cell` to a dummy resource request with identity `res:/dummy`. For my tests to succeed I therefore had to actually provide that dummy resource inside the target `<space>` of the mapper. I used a convenient recent feature of the standard module and used an inline `<literal>` resource declaration like this..

```
<mapper>
  <config>
    ...
  </config>
  <space>
    <literal uri="res:/dummy" type="string">ReplaceMe!</literal>
  </space>
</mapper>
```

I ran my tests. The second test - to `active:cell` - went green. We had a channel and it was returning a resource representation. No matter that it is a string with "ReplaceMe!" - we'd trimmed down the potential of the space and it was starting to gives us a representation for a cell.

Now it was simple to implement `c:x:y` all I needed to do was add this mapping...

```
<endpoint>
  <grammar>
    <simple>c:{x}:{y}</simple>
  </grammar>
  <request>
    <identifier>active:cell</identifier>
    <argument name="x">arg:x</argument>
    <argument name="y">arg:y</argument>
  </request>
</endpoint>
```

I cheated. I just made `c:x:y` an alias for a long-hand request to `active:cell`. If reading these mappings isn't second nature yet, then here's a visualizer trace of what's going on...

...notice that `c:o:o` gets mapped to `active:cell+x@o+y@o` and this gets mapped to the dummy resource. Notice also that my requests are all cacheable - this will start to come into play later.

At this point my first and second tests now passed.

Next I added a mapping for `active:cells`. It went green. Finally I added another alias

from cells{...} to active:cells. My last test went green.

My mapper config looked like this...

```
<config>
  <!--Cell Endpoints-->
  <endpoint>
    <grammar>
      <simple>c:{x}:{y}</simple>
    </grammar>
    <request>
      <identifier>active:cell</identifier>
      <argument name="x">arg:x</argument>
      <argument name="y">arg:y</argument>
    </request>
  </endpoint>
  <endpoint>
    <grammar>
      <active>
        <identifier>active:cell</identifier>
        <argument name="x" />
        <argument name="y" />
      </active>
    </grammar>
    <request>
      <identifier>res:/dummy</identifier>
    </request>
  </endpoint>
  <!--Cells{} Endpoints-->
  <endpoint>
    <grammar>
      <simple>cells\{{operand}}\}</simple>
    </grammar>
    <request>
      <identifier>active:cells</identifier>
      <argument name="operand">arg:operand</argument>
    </request>
  </endpoint>
  <endpoint>
    <grammar>
      <active>
        <identifier>active:cells</identifier>
        <argument name="operand" />
      </active>
    </grammar>
  </endpoint>
```

```

</grammar>
<request>
  <identifier>res:/dummy</identifier>
</request>
</endpoint>
</config>

```

Which, I think you'll agree starts to make the module.xml look much more complicated than it really is. So to clean up I moved these mapper configurations for the atomic resources to a separate file called atomMapperConfig.xml located in a directory org/netkernel/demo/tictactoe/atom/. I then pointed the mapper to this as a resource with an <import> and made sure this would resolve to my file by adding a <fileset> into the mapped <space>.

I ended up with my <rootspace> looking like this...

```

<rootspace>
  <mapper>
    <config>
      <import>res:/org/netkernel/demo/tictactoe/atom/atomMapperCon-
fig.xml</import>
    </config>
    <space>
      <literal type="string" uri="res:/dummy">ReplaceMe!</literal>
      <fileset>
        <regex>res:/org/netkernel/demo/tictactoe/.*</regex>
      </fileset>
    </space>
  </mapper>
</rootspace>

```

I checked my tests. All still green. I hadn't broken anything.

Checkpoint

You can download a snapshot of both my modules at this point here...

- [urn.org.netkernel.demo.tictactoe-1.1.1-r1.jar](http://urn.org/netkernel/demo/tictactoe-1.1.1-r1.jar)
- urn.test.org.netkernel.demo.tictactoe-1.1.1-r1.jar

Composite Resources

Using the same process, next I moved on to defining the resources row(y), column(x) and the two diagonals. Having just thought about the shorthand for c:x:y I didn't need to pause to think in order to adopt a similar shorthand notation for a row row:y and declared a test...

```
<test name="row:0">
  <request>
    <identifier>row:0</identifier>
  </request>
  <assert>
    <stringEquals>ReplaceMe!</stringEquals>
  </assert>
</test>
```

The test fails. So switching to the tictactoe <rootspace> I added a new endpoint to the mapper <config> with a grammar that would resolve row:x like this...

```
<config>
  <import>res:/org/netkernel/demo/tictactoe/atom/atomMapperConfig.xml</import>
  <endpoint>
    <grammar>
      <simple>row:{y}</simple>
    </grammar>
    <request>
      <identifier>cells{c:0:[[arg:y]],c:1:[[arg:y]],c:2:[[arg:y]]}</identifier>
    </request>
  </endpoint>
</config>
```

Notice I don't have to implement anything as we know from last time that row:x is actually just an alias to a set of cells. So all I had to do was declare the request for that set of cells using the shorthand cells{...} notation we'd already provided in the atom mappings. Notice that the mapper <config> allows you to mix resource imports with local <endpoint> mappings.

The tests all pass again.

Using cut and paste and renaming the things that needed renaming I added a test and a mapping for column:x and then both of the named sets of cells for diagonal:0 (top-left to bottom right) and for diagonal:1 (bottom-left to top-right).

My tests for atoms and composites are all passing. My space is starting to contain the resolvable resources we're interested in. (Don't worry about the representation's all being "ReplaceMe!" at this point the representation state is irrelevant).

Finally I tidy up my new composite resource mappings by moving them to a file called `compositeMapperConfig.xml` in directory next to the atom stuff. I changed the mapper to import this and ended up with a tidy module again...

```
<rootspace>
  <mapper>
    <config>
      <import>res:/org/netkernel/demo/tictactoe/atom/atomMapperCon-
fig.xml</import>
      <import>res:/org/netkernel/demo/tictactoe/composite/compositeMap-
perConfig.xml</import>
    </config>
    <space>
      <literal type="string" uri="res:/dummy">ReplaceMe!</literal>
      <fileset>
        <regex>res:/org/netkernel/demo/tictactoe/.*</regex>
      </fileset>
    </space>
  </mapper>
</rootspace>
```

My tests were green. I hadn't screwed the space up. At this point, I put down my tools and went off to reward myself with a cup of coffee. In thirty minutes I'd broken the back of the problem, in another 5 minutes I'd be done, with a full persistent cache-coherent implementation of the atom resource model and I wouldn't have to write a line of code ... which is where we'll start again next time...

Checkpoint

You can download a snapshot of the modules at this point here...

- [urn.org.netkernel.demo.tictactoe-1.1.1-r2.jar](http://urn.org/netkernel/demo/tictactoe-1.1.1-r2.jar)
- [urn.test.org.netkernel.demo.tictactoe-1.1.1-r2.jar](http://urn.test.org.netkernel/demo/tictactoe-1.1.1-r2.jar)

Alternate Perspective

Tom Geudens has risen to the challenge and has implemented an initial version of the core resource model...

<http://practical-netkernel.blogspot.be/2012/08/minimalistic-oxo.html>

This is really helpful since next time I'll be able to compare and contrast with the approach I'm heading towards.

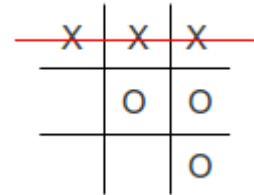
Incidentally, in Belgium they call tictactoe: "oxo" - same resource, different identifier. ♠

Resource Oriented Analysis and Design

Part 3

Peter Rodgers, PhD

Third in a series of articles taking a well understood domain problem and casting it as an ROC solution. Read Part 1 Here., Read Part 2 Here.



Recap

Last week we started defining the address space and set up logical endpoints (mapper mappings) for atomic and composite resources. Our tests were in place to probe the address space and ensure that it was configured correctly to resolve our requests. We deferred the problem of the representation state by mapping all requests to a dummy <literal> representation.

At the end of the article I closed with this cliff-hanger...

"My tests were green. I hadn't screwed the space up. At this point, I put down my tools and went off to reward myself with a cup of coffee. In thirty minutes I'd broken the back of the problem, in another 5 minutes I'd be done, with a full persistent cache-coherent implementation of the atom resource model and I wouldn't have to write a line of code ..."

So lets find out if I was full of BS or not?

Representation State Persistence

If you took the opportunity to look at [Tom's approach](#) you'll see that he solved the problem by constructing (the first C of the 3Cs) an endpoint as a Java class. It primarily consisted of a 2-D array data structure whose values would be set based on the position declared by his x,y arguments.

Aside from going with a more macroscopic solution (his atom is the entire array rather than each cell), he implemented the full range of verbs so that the state of the cells could be retrieved and or modified by SOURCE, EXISTS, SINK or DELETE requests. Nothing at all wrong with this.

He also recognises that the state of `active:cell` is naturally cacheable but that any state modifying request (SINK, DELETE) must cause the expiration of any previously cached representation. He adeptly uses the [golden thread pattern](#) (link is to localhost and assumes you're running the NK docs) to solve this, by associating all his SOURCE responses with the `gt:oxo` virtual resource like this..

```
//SOURCE and ATTACH GOLDEN THREAD
public void onSource(INKFRequestContext aContext) throws Exception {
    if (vMatrix == null) {
        throw new NKException("there is no game in progress");
    }

    INKFRequest vGTrequest =
aContext.createRequest("active:attachGoldenThread");
    vGTrequest.addArgument("id", "gt:oxo");
    aContext.issueRequest(vGTrequest);

    aContext.createResponseFrom(justForShow(vMatrix));
}

//DELETE and CUT GOLDEN THREAD
public void onDelete(INKFRequestContext aContext) throws Exception {
    if (vMatrix == null) {
        throw new NKException("there is no game in progress");
    }
    vMatrix = null;

    INKFRequest vGTrequest =
aContext.createRequest("active:cutGoldenThread");
    vGTrequest.addArgument("id", "gt:oxo");
    aContext.issueRequest(vGTrequest);

    aContext.createResponseFrom(true);
}
```

All in all, his first cut at the solution took about 130 lines of code. Not bad.

Here's how to do exactly the same thing using composition (the second C of the 3C's). We'll need two imports, a literal and have to change one mapping...

Compositional Solution

The problem of managing a simple state space of resources is a very common one. It doesn't matter what the problem domain is - it comes down to an identity for a resource which may be requested with SOURCE, SINK, DELETE or EXIST requests. It also, because this is ROC, doesn't matter what the representation of the resource is.

NetKernel provides out of the box a module which provides the Persistent Data Service (*pds:*) resource model. Its documentation is [here](#), but the essential point is that if you import the *pds* services your application space then has available the **pds:/** address space - which supports SOURCE, SINK, DELETE, EXISTS and which manages all the cache dependency and golden thread consistency for you.

Here's what I did.

First I import **urn:org:netkernel:mod:pds** - this space provides the high-level abstract *pds:/* service. I also imported **urn:org:netkernel:mod:pds:memory** - this space provides a basic in-memory persistence implementation which is called by the *pds:* service. (There is also a local RDBMS backed implementation and its pretty straight forward to implement your own PDS Persistence service with external RDBMS or NoSQL backends for distributed persistent state).

My mapped space now looked like this...

```
<space>
  <literal type="string" uri="res:/dummy">ReplaceMe!</literal>
  <fileset>
    <regex>res:/org/netkernel/demo/tictactoe/.*/</regex>
  </fileset>
  <!--Use PDS as our persistence service-->
  <import>
    <uri>urn:org:netkernel:mod:pds</uri>
  </import>
  <import>
    <uri>urn:org:netkernel:mod:pds:memory</uri>
  </import>
  <!--Literal PDS Config-->
  <literal type="xml" uri="res:/etc/pdsConfig.xml">
    <config>
      <zone>global:TicTacToe</zone>
    </config>
  </literal>
</space>
```

In the docs you'll see that the *pds:/* service always requires that the implementing space provide a contextual configuration resource called *res:/etc/pdsConfig.xml* (that is, *pds:* will always look for a local copy of *res:/etc/pdsConfig.xml* - just like on Unix, *ssh* will always request *~/.ssh/AuthorizedKeys* - its a well known pattern).

PDS uses the configuration resource to determine a collection name for this set of *pds*

resources (something that is called the "zone"). You can see that in my space I provided a very simple static <literal> implementation of res:/etc/pdsConfig.xml and declared my pds zone to be *gobal:TicTacToe* (remember this, we'll come back to it one day).

So now my space has available a full, extensible and consistent resource space. Now I just need to switch my endpoints from pointing to the dummy resource - to instead map into the pds: space. So I edited my atomMapperConfig.xml and changed the request from dummy to **pds:/ttt/cell/x/y** (where x and y are just relayed from the active:cell arguments)...

```
<endpoint>
  <verbs>SOURCE,SINK,DELETE,EXISTS</verbs>
  <grammar>
    <active>
      <identifier>active:cell</identifier>
      <argument name="x" />
      <argument name="y" />
    </active>
  </grammar>
  <request>
    <identifier>pds:/ttt/cell/[[arg:x]]/[[arg:y]]</identifier>
  </request>
</endpoint>
```

While I was at it, I also added a <verbs> declaration to tell the mapper that I want this mapping to respond to the full range of SOURCE, SINK, DELETE and EXISTS verbs which will simply be relayed on in the request to the *pds:* resource.

That's it. I had delivered a robust solution that was functionally equivalent to Tom's in exactly 45 seconds.

Stateful Testing

My next task was to go back to my tests and make sure that my composite solution was actually working. Here you can see that in a couple of minutes I modified my atom tests to do *setup*, *test* and *teardown*. The setup phase SINKs state to a *c:0:0*, the test then SOURCES that state (which is what we apply assertions to). The teardown then DELETE's the state - so that subsequent tests will start with a clean resource space...

```
<testlist>
  <test name="c:0:0">
    <setup>
      <verb>SINK</verb>
      <identifier>c:0:0</identifier>
```

```

    <argument name="primary">
      <literal type="string">X</literal>
    </argument>
  </setup>
  <request>
    <identifier>c:0:0</identifier>
  </request>
  <teardown>
    <verb>DELETE</verb>
    <identifier>c:0:0</identifier>
  </teardown>
  <assert>
    <stringEquals>X</stringEquals>
  </assert>
</test>
<test name="active:cell+x@0+y@0">
  <setup>
    <verb>SINK</verb>
    <identifier>c:0:0</identifier>
    <argument name="primary">
      <literal type="string">O</literal>
    </argument>
  </setup>
  <request>
    <identifier>active:cell</identifier>
    <argument name="x">0</argument>
    <argument name="y">0</argument>
  </request>
  <teardown>
    <verb>DELETE</verb>
    <identifier>c:0:0</identifier>
  </teardown>
  <assert>
    <stringEquals>O</stringEquals>
  </assert>
</test>

...

</testlist>

```

Executing the test and looking at the results I now see:

- c:0:0 test now returns "X"
- active:cell+x@0+y@0 test now returns "O"

Initialisation - ROC Fallback and Masking Patterns

OK this is starting to look good. But now the question arose: what about initialisation and setup? Tom has a simple solution, he made his stateful endpoint respond to NEW requests. But I had an elegant ROC-level pattern in mind. Why not imagine that all of the cells are already there in the space? If any given `c:x:y` cell had not yet received game state with a SINK - why not just fallback to a default empty cell?

Unsurprisingly this is also a common pattern for stateful address spaces. And *pds:* is already ahead of us. It implements a higher-order address space called *fpds:*. The semantics of which are: first try to SOURCE the resource in *pds:* if it is not there then *fallback* and make a request for the same identifier but in the generic *res:* space.

So to make my solution space be pre-initialised all I needed to do was change **pds** to **fpds** in my mapping like this...

```
<request>
  <identifier>fpds:/ttt/cell/[[arg:x]]/[[arg:y]]</identifier>
</request>
```

But then I also had to add the fallback **res:/ttt/cell/x/y** to my address space so that there would be a "platonic cell". So I also edited the `atomMapperConfig.xml` to provide a mapping to a default cell resource **res:/ttt/cell/default**.

```
<endpoint>
  <grammar>
    <simple>res:/ttt/cell/{x}/{y}</simple>
  </grammar>
  <request>
    <identifier>res:/ttt/cell/default</identifier>
  </request>
</endpoint>
```

Just one more thing - I had to then make sure this resource would resolve to a representation so I added another literal to my space...

```
<literal type="string" uri="res:/ttt/cell/default" />
```

Its a little strange looking (and maybe if I wasn't looking at the clock I'd do it differently) but hey, it returns an empty string "".

Finally I added a test that my expected initialisation default was working (notice there is no setup or teardown)...

```

<test name="active:cell+x@0+y@0 DEFAULT to res:/ttt/cell/default">
  <request>
    <identifier>active:cell</identifier>
    <argument name="x">0</argument>
    <argument name="y">0</argument>
  </request>
  <assert>
    <stringEquals />
  </assert>
</test>

```

Finally I also added a sequence of tests to make sure that SINKing was correctly going to pds: and that a subsequent DELETE would then return back to the default.

All in I'd added another two more minutes to my time sheet - but I had beaten the 5 minute target by 15 seconds. I went to reward myself with another cup of coffee...

Next time we'll move up to the composite resources...

Checkpoint

You can download a snapshot of the modules at this point here...

- urn.org.netkernel.demo.tictactoe-1.1.1-r3.jar
- urn.test.org.netkernel.demo.tictactoe-1.1.1-r3.jar

Note: Before you try these please make sure to update mod-pds to the latest version from apposite (shipped last week) - I discovered that the in-memory pds had a bug and did not reliably cut the internal golden thread on DELETE. ♠

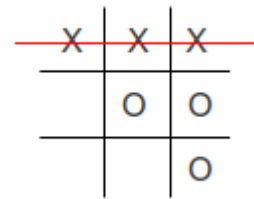
Resource Oriented Analysis and Design

Part 4

Peter Rodgers, PhD

Recap

Last week we implemented the stateful `active:cell` endpoint (and its shorthand alias `c:x:y`). We avoided writing any code by taking a compositional approach and mapping our resources to the `pds:` address space. We also discovered the potential offered by allowing the address space to implement "fallback" resources and used the `fpds:` variant of `pds:`.



Now we have to look at the composite resources and in particular how to implement `active:cells` and its alias `cells{...}`. But first we need to do a little groundwork...

Last time we saw that the `<setup>` and `<teardown>` requests in Xunit allow us to establish a known state for the context of the test request. With our focus on individual cells it was sufficient for us to setup by `SINKing` to a cell, `SOURCE` it in our test and `DELETE` its value in the teardown. But obviously this gets pretty tedious when we want to start working on the composite `active:cells` resource.

So our first job is to be as lazy as possible and find a way to make our life easy. In order to develop the `active:cells` resource we're going to need to try it out as we go - therefore before each attempt we need to be able to wipe out all cell state and restore all cells back to the default (platonic cell resource - remember last time?). So we need a state reset service.

A quick and easy first idea to solve this problem would be to implement an endpoint like this...

```
<endpoint>
  <verbs>DELETE</verbs>
  <grammar>
    <active>
      <identifier>active:cellResetState</identifier>
    </active>
  </grammar>
</request>
```

```

<identifier>active:groovy</identifier>
<argument name="operator">
  <literal type="string">

for(x=0;x<3;x++)
{
  for(y=0;y<3;y++)
  {
    context.delete("c:${x}:${y}".toString())
  }
}

  </literal>
</argument>
</request>
</endpoint>

```

Notice I'm doing the groovy code inline since its so short (I have the code in a CDATA section to make it readable but the XML serialization above has escaped the < and > characters).

...but as it happens I had bigger ideas and I didn't take this option. As it turns out I paid the price... here's the warts and all steps I took...

Implementing active:cellResetState

The reason I didn't go with the simple solution above, is that I wanted to be able to show you an ROC pattern that's pretty handy. I wanted to show how sometimes its useful to adopt a convention for our identifiers which allows us to talk about a collected set of resources, rather than single items.

I was pretty sure that pds: implemented this convention so I quickly implemented my answer to the reset with this endpoint declaration...

```

<endpoint>
  <verbs>DELETE</verbs>
  <grammar>
    <active>
      <identifier>active:cellResetState</identifier>
    </active>
  </grammar>
  <request>
    <identifier>pds:/ttt/cell/</identifier>
  </request>
</endpoint>

```

My expectation is that a DELETE request for active:cellResetState is mapped to a

DELETE request to pds:/ttt/cell/ which ought to allow us to delete multiple resources sharing the same root path. This is an instance of the naming convention I touched on above.

The convention is that an identifier ending with a trailing slash (/) is referring to all of the individual member resources below that path.

Deleteing everything in pds:/ttt/cell/ obviously covers the atomic set of cells we'd established last time when we implemented the atomic resources.

To check my new service I added this test ...

```
<test name= "DELETE active:cellResetState">
  <request>
    <identifier>active:cellResetState</identifier>
    <verb>DELETE</verb>
  </request>
  <assert>
    <true />
  </assert>
</test>
```

And naively, I expected that I had shown off another neat trick and avoided any code...
Oh such hubris, Oh such over-confidence; I was ripe for a fall...

```
<ex>
  <id>SubrequestException</id>
  <space>In Memory PDS Impl</space>
  <endpointId>PDSAccessorMemory</endpointId>
  <endpoint>PDSInMemoryImpl</endpoint>
  <ex>
    <id>java.lang.NullPointerException</id>
    <stack>
      <level>org.ten60.netkernel.mod.pds.PDSInMemoryImpl.delete()
line:109</level>
      <level>org.ten60.netkernel.mod.pds.PDSInMemoryImpl.onRequest()
line:58</level>
      <level>org.ten60.netkernel.mod.pds.PDSInMemoryImpl.onRequest()
line:35</level>
      <level>org.netkernel.layer0.nkf.impl.NKFEndpointImpl.onAsyncRequest()
line:93</level>
      <level>... 94 more</level>
    </stack>
  </ex>
</ex>
```

Aaaaagh! My test failed with a horrible exception. I clicked execute on the test to see the detailed stack trace above. NullPointerException - ouch. I'd found an edge in the ROC universe.

Since this was an NPE, I went to the mod:pds module, opened it up and looked at the source code for `org.ten60.netkernel.mod.pds.PDSInMemoryImpl.delete()`.

Ooops - on inspection I discovered that the pds in-memory impl is just a very simple in-memory key-value map - it doesn't implement the "trailing slash pattern" I was aiming to show off. Bugger!

[Note to the "not to be named author" of the in-memory impl - the NPE is not a nice way to behave - we should fix this!]

OK a quick face saving recovery is needed. I'm damned sure that pds: does implement this pattern as we use it all the time in the system tools. Its just we use the truly persistent version that is backed by a local RDBMS implementation.

What's it called? I typed "pds" into the search tool in the control panel and found the pds docs...

<http://localhost:1060/book/view/book:urn:org:ten60:netkernel:mod:pds/doc:mod:pds:title>














Ah ha that's what I want, I quickly switched my module's import to import the rdbms:local version rather than the in-memory version...

```
<import>
  <uri>urn:org:netkernel:mod:pds:rdbms:local</uri>
</import>
```

That should sort it. *Such conceit! My second fall was coming...*

I ran the unit tests to see if my new `active:resetCellState` endpoint was now fixed. At this point all hell broke loose...

/.

Date	Total Run	Success	Exceptions	Assert	Execution Time	Total Test Time
Thu, 13 Sep 2012 11:41:43 GMT	14	14	10	0	4	13
						122
Test	Time	Status				
1 c:0:0	1	assertionFailure				
		stringEquals				
		Expected: X				
		Received: org.netkernel.layer0.representation.ByteArrayRepresentation@70493388				
2 active:cell+x@0+y@0	1	assertionFailure				
		stringEquals				
		Expected: O				
		Received: org.netkernel.layer0.representation.ByteArrayRepresentation@2b5083ee				
3 active:cell+x@0+y@0 DEFAULT to res:/ttt/cell/default	1	success				
4 pds-fpds sequence 1 - c:0:0 SINK X	4	success				
5 pds-fpds sequence 2 - c:0:0 SOURCE equals X	2	assertionFailure				
		stringEquals				
		Expected: X				
		Received: org.netkernel.layer0.representation.ByteArrayRepresentation@7bb24966				
6 pds-fpds sequence 3 - c:0:0 DEFAULT back to res:/ttt/cell/default	1	success				
7 DELETE active:cellResetState	1	assertionFailure				
		true:				
8 active:cellResetState always resets to default cell	0	success				
9 cells{c:0:0,c:0:1}	0	success				
10 active:cells+operand@c:0:0,c:0:1	0	success				
11 column:0	0	success				
12 row:0	1	success				
13 diagonal:0	0	success				
14 diagonal:1	1	success				

At first glance bad. But relax and look again. Actually not too bad - since the tests are executing without errors its just that several assertions are failing...

Look at one of the assertions and think a moment...

Ah look at that! Its the `<stringEquals>` assert that's failing. Hmmmm, because I've switched the pds: implementation it's now returning a `ByteArrayRepresentation` instead of a `java.lang.String`.

Oh of course - the in-memory pds impl is a POJO-bucket that will hold any old POJO representation, but the RDBMS-backed version of pds is a *bit-bucket* and necessarily has to store its state as Blobs. So it always requires its argument resources as binary streams. When we SOURCE from it without expressing any representation type preference we get serialized binary streams back!

In order to be equivalent to what I had before (String POJOs as my basic representation model) I needed to ensure that my request to pds: automatically transrepted the representation to a `java.lang.String`. So looking at my `atomMapperConfig.xml` I modified the request to pds: and added a `<representation>` declaration to say "Don't come back if you're not a

String"...

```
<request>
  <identifier>fpds:/ttt/cell/[[arg:x]]/[[arg:y]]</identifier>
  <representation>java.lang.String</representation>
</request>
```

That should sort it! Wrong again. This time when I ran the tests I saw exceptions! Clicking on a failing test I saw...

```
Request Resolution Failure
TRANSREPT from ByteArrayRepresentation to String
```

Primitive type transreption errors like this happen when your address space doesn't have any primitive type transreptors (not a very helpful sentence - but its a truism). Oh yeah of course, to allow my endpoints to automatically discover a transreption pipeline I'd forgotten to import `layer1` into my mapper's space so that it had access to all the low-level transreptors that `layer1` offers...

```
<import>
  <uri>urn:org:netkernel:ext:layer1</uri>
</import>
```

OK all back in the green again - I had fully reimplemented my persistence layer (now with true long-term persistence not transient in-memory) and I was confident that the typing was all correct and that my high level tictactoe atomic resource model was behaving exactly the same as before...

All except my damned active:cellResetState test I'd added! I was asserting `<true>`, expecting the DELETE on `pds:/ttt/cell/` to succeed - but it was coming back false.

Damn Damn Damn... should I give up and just implement the simple groovy cell deletion script after all? I could just pretend like I never made this misstep - dear reader, you would be none-the-wiser.

No way. That would be a lie and a defeat and... and... and... I'd have had to write some code!!

So with a cry of "No surrender" - which startled the others in the office. I dug deep and looked closely at the atomic `active:cell` mapping to `fpds:.` The mapped request looked like this...

```
<request>
  <identifier>fpds:/ttt/cell/[[arg:x]][[arg:y]]</identifier>
  <representation>java.lang.String</representation>
</request>
```

Ah ha, a little subtle, but look at that. I hadn't given any thought to the actual identity of the pds: path I'd used when I set this up. Could it be the slash delimited path notation `fpds:/ttt/cell/x/y` that is causing the problem?

We are trying to delete set `pds:/ttt/cell/` but I seem to remember that the pds: model doesn't differentiate super-sets - its a bit like the unix-path and expects the trailing set convention to only apply to members after the last slash. (Err its in the name "trailing slash convention").

So, for efficiency of implementation, the pds-local-RDMBS impl only thinks of the last slash as being the identity of a collection of resources, not the higher level paths. So to succeed what I really needed was to ensure that all my mapped cells went into a single pds: "trailing slash collection set" like this: `pds:/ttt/cell/x-y`.

Oh thats easy! I can just change the mapping. Lets get rid of the slash delimiter between x and y and instead lets use a dash (-)...

```
<request>
  <identifier>fpds:/ttt/cell/[[arg:x]]-[[arg:y]]</identifier>
  <representation>java.lang.String</representation>
</request>
```

I also remembered that because we're using fpds: I also had to change the fallback resource too.

```
<endpoint>
  <grammar>
    <simple>res:/ttt/cell/{x}-{y}</simple>
  </grammar>
  <request>
    <identifier>res:/ttt/cell/default</identifier>
  </request>
</endpoint>
```

I make the changes. Run the tests... and ... we're all green! Yes! By the skin of my teeth I'd avoided writing any code.

And who knows - this stupid little detour might have cast some light on the trailing-slash pattern and also revealed some implementation detail on pds:. Maybe not a wasted journey. Anyway this had taken about 5 minutes - so not too bad.

OK lets make sure we're actually resetting the state by adding a test with some sequences of stateful requests. Here's a sequence of tests that first SINKs, then resets and then SOURCES to show that our active:cellResetState endpoint is working...

```
<testlist>
  <test name="active:cellResetState - prepare state of c:0:0 and don't cleanup">
    <request>
      <verb>SINK</verb>
      <identifier>c:0:0</identifier>
      <argument name="primary">
        <literal type="string">X</literal>
      </argument>
    </request>
  </test>
  <test name="DELETE active:cellResetState - cleanup">
    <request>
      <verb>DELETE</verb>
      <identifier>active:cellResetState</identifier>
    </request>
    <assert>
      <true />
    </assert>
  </test>
  <test name="c:0:0 should now have been reset to default cell">
    <request>
      <identifier>c:0:0</identifier>
    </request>
    <assert>
      <stringEquals />
    </assert>
  </test>
</testlist>
```

This sequence of tests are all green. active:cellResetState works! Now we can reset the state easily. Now we can actually start on the real story this week...

Implementing active:cells

You will recall that the composite resource **active:cells** was left as a simple mapping to the dummy resource which when requested in our tests is returning "ReplaceMe!"...


```

<endpoint>
  <grammar>
    <active>
      <identifier>active:cells</identifier>
      <argument name="operand" />
    </active>
  </grammar>
</request>
  <identifier>res:/dummy</identifier>
</request>
</endpoint>

```

It follows that all of the other composite resource mappings such as `row:y` and `column:x` are producing the same state (since they're all ultimately just aliases to `active:cells`).

Time to make `active:cells` do something real. Dare I say it, time for some code!

I changed the request mapping from `res:/dummy` to `active:groovy` like this...

```

<request>
  <identifier>active:groovy</identifier>
  <argument name="opera-
tor">res:/org/netkernel/demo/tictactoe/atom/cellsImpl.gy</argument>
  <argument name="operand">arg:operand</argument>
</request>

```

I also, having had my fingers burned above by forgetting to provide imports when introducing new services, added an import of the `groovy` module `urn:org:netkernel:lang:groovy` into my mapper space so that `active:groovy` would be resolvable.

Finally I created a script `cellsImpl.gy` in the `atom/` directory. To start with, I made it implement a code-based equivalent to the dummy mapping - as you can see, I returned a string telling me that I hadn't done anything yet...

```
context.createResponseFrom("Not Finished Yet")
```

I ran my tests to make sure that the mapping was correct and that my code was running instead of the dummy resource...

10	cells{c:0:0,c:0:1}	14	assertionFailure	exec
			stringEquals	
			Expected: ReplaceMe!	
			Received: Not Finished Yet	

This time the assert failure is giving me some good news. Its saying that my test is calling the code since its now returning "Not Finished Yet" but unfortunately all my asserts for the composite resources are expecting the old dummy resource.

So, using the power of search-replace, I did a global search for "Replace Me!" and changed it to "Not Finished Yet"... now, all the composite resource asserts looked like...

```
<assert>
  <stringEquals>Not Finished Yet</stringEquals>
</assert>
```

Rerun the tests and all is good. In fact all is *exceptionally* good. Something rather pleasing just happened. I changed all the asserts including all the asserts on the `row:y`, `column:x` and `diagonal:z` tests. They're all passing too.

What this is telling me is that all of the composite services are all correctly mapping to the `active:cells` endpoint and its single line of implementation code. My new dynamic implementation of `active:cells` is now being used by *all* of the composite resources that map to it. I have a single normalised focal point to concentrate my efforts on and I know they will be reflected into all the other services...

OK we're ready to implement a real `active:cells` solution. But I need to focus and check my progress so I choose the first `active:cells` test and execute it (test 10 in the list)...

<http://localhost:1060/test/exec/html/urn:test.org:netkernel:demo:tictactoe?index=10>

I see "Not Finished Yet" in my browser. Now I can change the code and refresh this link to iteratively solve the problem.

First off I add another line of code to echo back the operand argument (you'll recall this is the composite identifier containing the list of cell identifiers)...

```
operand=context.getThisRequest().getArgumentValue("operand")
context.createResponseFrom(operand)
```

When I F5-refresh my browser pane I get back...

```
c:0:0,c:0:1
```

OK I can see the cells. Next we need to split this on the comma separator "," to get a list of identifiers...

```

operand=context.getThisRequest().getArgumentValue("operand")
cells=operand.split(",");
for(i=0; i<cells.size(); i++)
{
    println(cells[i])
}
context.createResponseFrom(operand)

```

I println to stdout and glance at the console to see that I have the identifiers being spat out. OK nearly done. My final job is to build an HDS tree representation...

```

import org.netkernel.layer0.representation.*
import org.netkernel.layer0.representation.impl.*;

operand=context.getThisRequest().getArgumentValue("operand")
cells=operand.split(",");
b=new HDSBuilder();
b.pushNode("cells")
for(i=0; i<cells.size(); i++)
{
    b.addNode("cell",cells[i]);
}
context.createResponseFrom(b.getRoot())

```

Now in my browser refresh I get...

```

<cells>
  <cell>c:0:0</cell>
  <cell>c:0:1</cell>
</cells>

```

Which is just a structured list of cell identifiers! I need to dereference these identifiers. By which I mean I need the representation state of the identified cell. So I need to SOURCE each identifier. Here's the final result...

```

import org.netkernel.layer0.representation.*
import org.netkernel.layer0.representation.impl.*;

operand=context.getThisRequest().getArgumentValue("operand")
cells=operand.split(",");
b=new HDSBuilder();
b.pushNode("cells")
for(i=0; i<cells.size(); i++)
{
    b.pushNode("cell",context.source(cells[i]));
    b.addNode("@id", cells[i])
    b.popNode();
}

```

```
context.createResponseFrom(b.getRoot())
```

And in my browser I see...

```
<cells>
  <cell id="c:0:0" />
  <cell id="c:0:1" />
</cells>
```

The important detail is that I added `context.source(cell[i])` which in this example means I am making requests for `c:0:0` and `c:0:1` and incorporating the representation state into the HDS tree structure. Notice also that since I know the identity of the cell resource - I might as well retain it for future reference so I add it as an attribute on the cell node.

Progress. But this is now not a good test since it is just pointing to a set of empty cells. I need to modify my test to setup some initial state and check that my active:cells implementation is really doing what I think...

```
<test name="cells{c:0:0,c:0:1}">
  <setup>
    <verb>SINK</verb>
    <identifier>c:0:0</identifier>
    <argument name="primary">
      <literal type="string">X</literal>
    </argument>
  </setup>
  <request>
    <identifier>cells{c:0:0,c:0:1}</identifier>
  </request>
  <teardown>
    <verb>DELETE</verb>
    <identifier>active:cellResetState</identifier>
  </teardown>
  <assert>
    <xpath>/cells/cell[1]='X'</xpath>
  </assert>
</test>
```

The test now gives me...

```
<cells>
  <cell id="c:0:0">X</cell>
  <cell id="c:0:1" />
</cells>
```

I have a real solution. `active:cells` is finished in 12 lines of code.

One thing to point out in my updated test. Notice that the assert has changed - its now using an `<xpath>` assert. Xpath is a great way to give a rich assertion on tree structured data. In this case I'm saying: the first cell should have the value "X".

But to be able to use the `<xpath>` assert I need to edit my Xunit `testlist.xml` and import the xml assertion library. I can never remember what its called so I search "xml assert" in the docs. The page comes up and so I add the import to my `testlist.xml`...

```
<import>res:/org/netkernel/xml/assert/assertLibrary.xml</import>
```

Again I remember that my test space (in my test module) knows nothing about the `xml:core` library (which provides the xml assertion services) so therefore I need to add an import for `urn:org:netkernel:xml:core`.

My first test of `active:cells` is now complete and showing that I have a full working implementation. But of course, since my composite resource tests are live and hitting the `active:cells` endpoint they are all showing assert *failures* since they are still using `<stringEquals>` asserts. I quickly do a search-replace...

```
<stringEquals>Not Finished Yet</stringEquals>
```

is replaced with...

```
<xpath>/cells</xpath>
```

Hey presto all my `row:y`, `column:x` and `diagonal:z` resources start passing their tests and so must be producing real `<cells>` resources!

Finally and for good measure I `<setup>` some initial state for each of those tests so that I can prove they're all for real.

I've finished the problem. Total time (including the stupid state reset detour) 15 minutes.

Board Yet?

I was still highly caffeinated from my earlier relaxation breaks. So decided to press on a little further. I knew that Tom had implemented a way to get back a representation of the state of the complete board. What would it take for me to do the same? Perhaps more im-

portantly, could I do it with declarative composition and no more lines of code? (The twelve lines in the `active:cells` implementation was already pushing my limit for a working day).

Of course I already knew the answer... I quickly created a file called `board-hr1.xml` in the `composite/` directory. With a bit of cut and paste it looked like this...

```
<board>
  <row>
    <request>
      <identifier>row:0</identifier>
    </request>
  </row>
  <row>
    <request>
      <identifier>row:1</identifier>
    </request>
  </row>
  <row>
    <request>
      <identifier>row:2</identifier>
    </request>
  </row>
</board>
```

I created the basic structure and one `<row>` then copied and pasted the others then I changed the identifier in the row `<request>` to point to the correct resource I wanted.

Next I added a new `active:board` endpoint using a mapping in the `compositeMapper-Config.xml` which maps to `active:hr1` and tells it to evaluate the `board-hr1.xml` resource...

```
<endpoint>
  <grammar>
    <simple>active:board</simple>
  </grammar>
  <request>
    <identifier>active:hr1</identifier>
    <argument name="operator">res:/org/netkernel/demo/tictactoe/composite/board-hr1.xml</argument>
  </request>
</endpoint>
```

`active:hr1` is a recursive HDS composition language - it is a member of the family of recursive composition languages which includes XRL and TRL. It is very simple. It recursively evaluates embedded requests and adds the HDS representation state to that location in the tree structure. It supports declarative requests (used above) or full NKF request objects. It is also trivial to make the evaluated requests asynchronous by adding `@async` attribute to the

declarative request.

I knew that HRL is in a library so I added an import to my mapper space for `urn:org:netkernel:lang:hrl` (if its not installed on your machine look for `lang-hrl` in ap-posite). You can see how simple it is from the docs here

<http://docs.netkernel.org/book/view/book:lang:hrl:book/doc:lang:hrl:title>

Finally I added a test...

```
<test name="active:board">
  <setup>
    <verb>SINK</verb>
    <identifier>c:1:1</identifier>
    <argument name="primary">
      <literal type="string">X</literal>
    </argument>
  </setup>
  <request>
    <identifier>active:board</identifier>
  </request>
  <teardown>
    <verb>DELETE</verb>
    <identifier>active:cellResetState</identifier>
  </teardown>
  <assert>
    <xpath>count (/board/row)=3</xpath>
    <notExpired />
    <minTime>5</minTime>
  </assert>
</test>
```

I ran the test and got back...

```
<board>
  <row>
    <cells>
      <cell id="c:0:0" />
      <cell id="c:1:0" />
      <cell id="c:2:0" />
    </cells>
  </row>
  <row>
    <cells>
      <cell id="c:0:1" />
      <cell id="c:1:1">X</cell>
```

```

        <cell id="c:2:1" />
    </cells>
</row>
<row>
    <cells>
        <cell id="c:0:2" />
        <cell id="c:1:2" />
        <cell id="c:2:2" />
    </cells>
</row>
</board>

```

Job done. Literally this took one minute.

I wanted to show one more thing. I cut and pasted a copy of my `active:board` test. So I now had two of them back to back. I tweaked the first to have the setup and the second to do the teardown - since I wanted my requests to `active:board` to run on exactly the same cell state...

```

<testlist>
  <test name="active:board">
    <setup>
      <verb>SINK</verb>
      <identifier>c:1:1</identifier>
      <argument name="primary">
        <literal type="string">X</literal>
      </argument>
    </setup>
    <request>
      <identifier>active:board</identifier>
    </request>
    <assert>
      <xpath>count(/board/row)=3</xpath>
      <notExpired />
      <minTime>5</minTime>
    </assert>
  </test>
  <test name="active:board - cache test">
    <request>
      <identifier>active:board</identifier>
    </request>
    <teardown>
      <verb>DELETE</verb>
      <identifier>active:cellResetState</identifier>
    </teardown>
    <assert>

```



```

    <xpath>count (/board/row)=3</xpath>
    <notExpired />
    <maxTime>0</maxTime>
  </assert>
</test>
</testlist>

```

Take a close look at the asserts. In the first I am saying "I expect this to take at least 5ms". In the second I am saying "I expect this to take 0ms". In both cases I expect the representation to be `<notExpired>`.

Aside from the ease of being able to create composite resources, as we see in the `row:y`, `column:x` and `active:board` solutions. Here at last, is the evidence of a further massive pay-back for decomposing the state into atomic and composite resources.



The composite resource `active:board` has requests for `row:x` which maps to `active:cells` which has requests for `active:cell`. But what we're seeing is that every single resource starts to roll up into the high-level composite.

So long as nothing changes *nothing* needs to be recomputed.

If something does change (like one cell when we start playing the game) only one `active:cells` request (and therefore its alias `row:x`) are affected. Therefore when the `active:board` is requested again we automatically recompute only the affected row and, it follows, `active:cells` only requires that the one affected cell is actually requested directly. Everything else is the same as it was last time.

We have achieved a *completely normalized computation* for the state of the tictactoe game.

And the visualizer confirms this. Here's a filtered view of my tests showing just the back-to-back `active:board` requests...

Filter: active:board							
Physical Endpoint / Callstack	Space	Verb	Request Identifier	Duration (ms)	CPU (ms)	Cache	Response Representation
MapperOverlay	Demo / Tic Tac Toe / rootspace	SOURCE	active:board	26	0.02		HDS
[from cache]		SOURCE	active:board	0	0.00		HDS

Alternative Board with Code

Just for the hell of it I implemented a code-based solution to the same problem. I added a mapping for `active:boardWithCode` to `active:groovy` and the following script...

```
import org.netkernel.layer0.representation.*
```

```

import org.netkernel.layer0.representation.impl.*;

b=new HDSBuilder();
b.pushNode("board")
for(i=0; i<3; i++)
{
    b.pushNode("row")
    row=context.source("row:"+i, IHDSNode.class);
    b.importNode(row.getFirstNode("/cells"));
    b.popNode();
}
context.createResponseFrom(b.getRoot())

```

Not much code but just compare it with the compositional approach. It has so many moving parts. So many ways it could break or be misinterpreted or become brittle when required to add new features in response to change.

The HDSBuilder is convenient but it separates me from *thinking in the domain of the solution*. While the HRL composition actually lets me inspect and play with the structure of the result in the same domain (just by copying and pasting). I literally didn't need to consciously think how to solve the problem I just "painted the structure I wanted". But the composite approach goes much further...

It makes it trivially simple for me to augment it. What if, as we get into developing the actual game, I need to return the history in the same resource as the board's current state? Simple, I would just embed a request to active:history (which we haven't started on yet). What if we're scaling this service up to run as a stupid facebook app playing a 100 million simultaneous games. We would surely want to parallelize the requests to the persistence engine to amortize the network transfer time? No problem, just add an @async attribute - much much easier than dealing with callbacks or iterating over asynchronous NKF request handles.

What's with the Code Aversion?

What I hope I'm starting to show, is that our instinctual tendency "to start with code" is not always a good idea. I hope that it's starting to become apparent that declarative composition can produce extremely concise solutions and yet offer *scale-free evolvability*.

Of course, and as Tony keeps pointing out, when I say I did it with "no code". What I really mean is that I have minimised my need for "imperative code". I have, as much as possible, stayed in the declarative compositional world.

And you might ask yourself: *Do I know a wildly successful technology which adopts this same declarative compositional model?*

And then look at what you're using to view this story? Yep, all I'm doing inside my ROC solution is using the self-same patterns and proven economics that your web-browser uses.

Ask yourself: *Would web-page development be easier with imperative code or with declarative HTML?*

So why do we feel the need to start with code to solve general information (software) problems?

Next time we'll look at implementing the game and find there's not a lot to it...

Checkpoint

You can download a snapshot of the modules at this point here...

- `urn.org.netkernel.demo.tictactoe-1.1.1-r4.jar`
- `urn.test.org.netkernel.demo.tictactoe-1.1.1-r4.jar`

Note: You will need to install lang-hrl from apposite if you don't already have it. ♠

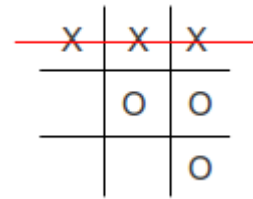
Resource Oriented Analysis and Design

Part 5

Peter Rodgers, PhD

Recap

Last week we implemented persistence of the cell state. We also added an implementation of the higher level composite resource active:cells endpoint and used a recursive "linked data" composition pattern.



Having established the core state of the TicTacToe resource model, today we need start to think about the game itself...

Working from Both Edges towards the Middle

When building a solution I like to follow a very simple mental model. It goes like this...

1. What have I promised that I will provide?
2. What do I need in order deliver my promise?
3. Add value (the thing that makes this special)
4. Return a response

It turns out this sequence of steps is scale invariant. That is, the same steps apply when you are locally implementing the internals of an accessor as when you're implementing a complete channel through a top-to-bottom multi-tier architecture.

Essentially this is a recipe for local success and it implicitly suggests a "directed spacial relationship" to the contextual architecture at each step.

- Step 1 is focused on looking upwards to any layer above.
- Step 2 is saying what arguments have I been given but, also, what further resources should be requested - ie what should I expect from promises others have made to me (from peers at the same level or often from other layers below)
- Step 3 is what makes this endpoint unique.
- Step 4 just says, you should always keep your promise.

It follows that these steps suggest that a good working practice is to try to work from the edges inwards - since then you'll know what you've promised and what's been promised to you.

This contrasts with Object Oriented software, in which the focus on object design and relationships tends to start in the middle and move to both edges. Hence the term "middle-ware".

By working at the edges first - our hope, and the ideal, is that there is no middle. I sometimes think of this as "no-ware".

So lets find out how much middle we really need...

WWW Surface

Our unit test fixtures gave us an easy way to probe the resources of the core model. But now we're working on the application itself we want a more rich and dynamic interactive development model.

We need to see instant results to our changes. We want a short virtuous circle of feedback.

Iterative feedback provides the environment for creativity and inspiration. Imagine trying to paint a picture if you only looked at it after you'd covered all the canvas? Creativity is a process. It requires you to adapt and change course when a new idea surfaces.

So the first job is to create a new module. Somewhere we can work on a live web rendering layer. Which is another way of saying - we need a top-edge to complement the bottom-edge of the TicTacToe resource model.

Using cut and paste of the boiler plate, like we did in part 2, I quickly knocked up a new module called `urn:org:netkernel:demo:tictactoe:www`.

I knew that I'd need a mapper so set one up pointing to an empty configuration for the mappings (`wwwMapperConfig.xml`). I added a `SimpleDynamicImportHook.xml` resource to make sure this module would get imported into the HTTPFulcrum and hence be exposed to Web HTTP requests. Finally I added an import of the `tictactoe` resource model `urn:org:netkernel:demo:tictactoe` (the bottom edge to which we'll work towards) ...

```
<module version="2.0">
  <meta>
    <identity>
      <uri>urn:org:netkernel:demo:tictactoe:www</uri>
```

```

    <version>1.1.1</version>
  </identity>
  <info>
    <name>Demo / Tic Tac Toe / WWW</name>
    <description>Tic Tac Toe Web Application</description>
  </info>
</meta>
<system>
  <dynamic />
</system>
<rootspace>
  <fileset>
    <regex>res:/etc/system/SimpleDynamicImportHook.xml</regex>
  </fileset>
  <mapper>
    <config>
      <import>res:/resources/www/wwwMapperConfig.xml</import>
    </config>
    <space>
      <fileset>
        <regex>res:/resources/www/.*</regex>
      </fileset>
      <import>
        <uri>urn:org:netkernel:demo:tictactoe</uri>
      </import>
    </space>
  </mapper>
</rootspace>
</module>

```

I added the new module to my `etc/modules.d/tictactoe.xml` file and it was discovered and commissioned.

In order that I could start to see something happening, I quickly added a mapping from `res:/demo/tictactoe/` to `active:board` in the file `wwwMapperConfig.xml`...

```

<config>
  <endpoint>
    <grammar>res:/demo/tictactoe/</grammar>
    <request>

      <identifier>active:board</identifier>
    </request>
  </endpoint>
</config>

```

In my browser I clicked this link...

<http://localhost:8080/demo/tictactoe/>

And lo and behold I saw an empty <board> representation - just like we saw in our `active:board` unit test (see last week if you don't remember it).

So I now had a top-edge connected to the bottom-edge and no middle. Here's what my **spacial architecture** looks like... *(in some pdf readers you can rotate and magnify the image below for better viewing)*

Giving it some style

So far this raw representation rendering was not very inspiring. So my next step was to make it look like a minimal tictactoe game. I changed the mapping to this...

```
<endpoint>
  <grammar>res:/demo/tictactoe/</grammar>
  <request>
    <identifier>active:xslt</identifier>
    <argument name="operand">active:board</argument>
    <argument name="operator">res:/resources/www/xslt/styleBoard.xsl</argument>
  </request>
  <header name="mime">text/html</header>
</endpoint>
```

And wrote a hacky stylesheet to turn the data into an HTML table representation of the board - don't worry about the ugly approach - I just cut and paste the structure and changed the paths - at this point we're not after perfection - we just want to see something...

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
  <xsl:output method="xml" />
  <xsl:template match="/board">
    <div>
      <style type="text/css">
        .cell {padding: 5px; width: 50px; height: 50px; background-color:
white;}
      </style>
      <table cellpadding="0" cellspacing="2" style="background-color: black; padding: 0px;">
        <tr>
          <td class="cell" id="c:0:0">
            <xsl:value-of select="row[1]/cell[1]" />
          </td>
          <td class="cell" id="c:1:0">
            <xsl:value-of select="row[1]/cell[2]" />
          </td>
          <td class="cell" id="c:2:0">
            <xsl:value-of select="row[1]/cell[3]" />
          </td>
        </tr>
        <tr>
          <td class="cell" id="c:0:1">
            <xsl:value-of select="row[2]/cell[1]" />
          </td>
          <td class="cell" id="c:1:1">
```



```

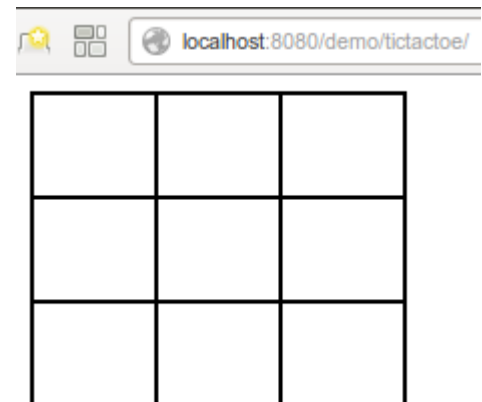
        <xsl:value-of select="row[2]/cell[2]" />
    </td>
    <td class="cell" id="c:2:1">
        <xsl:value-of select="row[2]/cell[3]" />
    </td>
</tr>
<tr>
    <td class="cell" id="c:0:2">
        <xsl:value-of select="row[3]/cell[1]" />
    </td>
    <td class="cell" id="c:1:2">
        <xsl:value-of select="row[3]/cell[2]" />
    </td>
    <td class="cell" id="c:2:2">
        <xsl:value-of select="row[3]/cell[3]" />
    </td>
</tr>
</table>
</div>
</xsl:template>
</xsl:stylesheet>

```

When it failed first time - I remembered that yet again I'd forgotten to add an import to my mapper space to provide `active:xslt` - so I added an import for `urn:org:netkernel:xml:core`. When I clicked the link again I saw this...

(From the stylesheet you can see that I am not trying to make this a well formed HTML page - its just a `<div>` fragment. But I'm getting my browser to render it nicely by ensuring that I return a mimetype of "text/html" - look at the mapper definition above, notice that I'm adding an explicit "mime" `<header>`).

Progress - its starting to look like the game. But of course the cells are empty as the model has no state.



Checkpoint

You can download a snapshot of the www module at this point here...

- `urn.org.netkernel.demo.tictactoe.www-1.1.1-r1.jar`

Adding Structured

So while I had a cheap and dirty representation of the board, it was, not to mince words - a heap of crap. A context free div fragment is not a web application. I needed a contextual page. No problem I'll use the recursive composition pattern again to create a recursive template structure.

I copied and pasted my endpoint. I renamed the grammar for the active:xslt request to be `res:/demo/tictactoe/board` - which is a more specific name. I then mapped the original `res:/demo/tictactoe/` to a request to `active:xrl2`.

XRL is a recursive XML resource composition language - in the same family as active:hds. It requires a *template* argument which will be recursed. In this case I also provided a content argument (my name) and provided a reference to the XSLT generated board resource...

```
<config>
  <endpoint>
    <grammar>res:/demo/tictactoe/</grammar>
    <request>
      <identifier>active:xrl2</identifier>
      <argument name="template">res:/resources/www/template/main.xml</argument>
      <argument name="content">res:/demo/tictactoe/board</argument>
    </request>
    <header name="mime">text/html</header>
  </endpoint>
  <endpoint>
    <grammar>res:/demo/tictactoe/board</grammar>
    <request>
      <identifier>active:xslt</identifier>
      <argument name="operand">active:board</argument>
      <argument name="operator">res:/resources/www/xslt/styleBoard.xsl</argument>
    </request>
  </endpoint>
</config>
```

I created the necessary directory structure and implemented `main.xml` like this...

```
<html xmlns:xrl="http://netkernel.org/xrl">
  <head>
    <script src="http://localhost:1060/nkse/style/js/jquery.js"
      type="text/javascript">_</script>
  </head>
  <body>
```

```

<h1>TicTacToe Demo</h1>
<div id="board">
  <xrl:include>
    <xrl:identifier>arg:content</xrl:identifier>
  </xrl:include>
</div>
</body>
</html>

```

Notice the embedded `<xrl:include>` - this is a declarative request that is evaluated by the XRL2 runtime. Here it will source the content argument - which as we saw in the declaration will be dereferenced to `res:/demo/tictactoe/board` - which in turn will invoke the active:xslt styling of the active:board resource.

While I was at it, do you notice I imported into the page the JQuery library (I know this is in the backend-fulcrum - so for now I'll just reference its URL on the other transport - we can sort this out later).

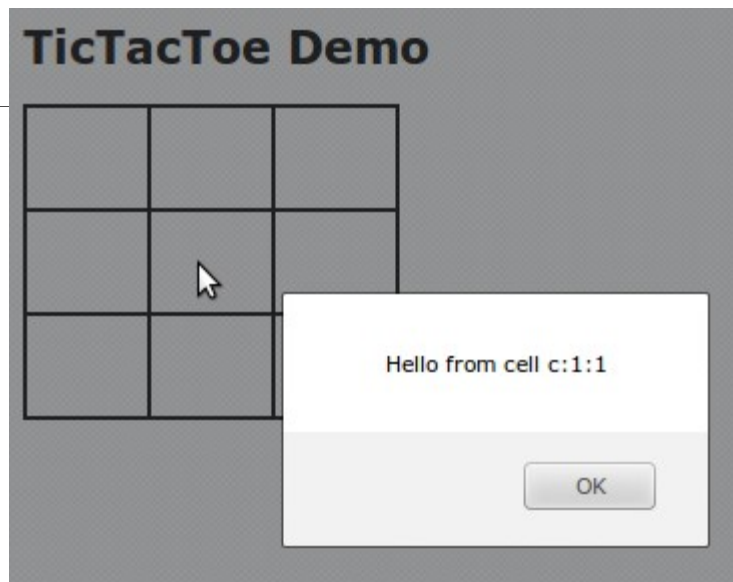
So now I can animate my solution. I added the following snippet of javascript. I actually placed this inside the XSLT template since its where my focus of attention is and I need to be able remember the structure of the HTML table for the board. (I can always move it later)...

```

<script type="text/javascript">
  <!--
function bindCell()
{
  //alert("Binding");
  $(".cell").click(
    function()
    { var cell=$(this);
      alert("Hello from cell "+cell.attr("id"));
    }
  );
}
bindCell();
-->
</script>

```

If you don't know JQuery the gist of it is this - when this div fragment is rendered the function `bind-Cell()` will be invoked. This function finds all elements with a class of "cell" (see my table stylesheet). For each discovered element bind the



"click" event to invoke the inline anonymous function. For now I show that things are working by displaying an alert box with a message.

I go back to my browser and refresh. The table looks the same - but now when I click a cell I see this...

Result. We're pretty much there already.

Checkpoint

You can download a snapshot of the www module at this point here...

- [urn.org.netkernel.demo.tictactoe.www-1.1.1-r2.jar](#)

Restart at the Beginning

Now I remembered from my unit testing of the core resource model that its important to be able to cleanly reset the state. So I added a link to my main.xml template like this.

```
<div style="margin-top: 50px;">
  <a href="restart">[Restart]</a>
</div>
```

Its going to request the relative URL "restart". So now I have to map that to something. I quickly copy and paste an endpoint definition and write an inline groovy script like this...

```
<endpoint>
  <grammar>res:/demo/tictactoe/restart</grammar>
  <request>
    <identifier>active:groovy</identifier>
    <argument name="operator">
      <literal type="string">
        context.delete("active:cellResetState")
        resp=context.createResponseFrom("Reset")
        resp.setHeader("httpResponse:/redirect", "./")
        resp.setExpiry(resp.EXPIRY_ALWAYS)
      </literal>
    </argument>
  </request>
</endpoint>
```

Just like in our unit tests cleanup, it simply makes a DELETE request on active:cellResetState. I'm doing a funky trick known to all HTML-jockeys. The browser will request "<http://localhost:8080/demo/tictactoe/restart>" - but I don't want it stay there so my re-

sponse includes a relative redirect back to the original /demo/titactoe/ path. Result I have an invisible reset/redirect/re-render of the page.

In my browser I try the Reset link and use the visualizer to see the correct series of requests going down from the top-edge to the bottom. I have a reset.

Putting down some moves

OK we're now ready to take first cut at implementing a real game. I can't quite remember the exact order that I did the next steps but here's a retrospective view...

The first rule of "no-ware" is to delegate state management to the edges. That is you don't want state in the middle since it is very brittle and is a scaling bottleneck.

I followed this rule and added a variable called "player" to the javascript of the main template. This will hold the token (X or O) of the currently active player. I also add a function togglePlayer to toggle the token value.

The script in main.xml looks like this...

```
<script type="text/javascript">
  <!--
var player="X";
function togglePlayer()
{  if(player=="X")
    {  player="O";
    }
    else
    {  player="X";
    }
}
function refreshBoard()
{  $("#board").load("board");
}
-->
</script>
```

Notice that I also added a function to do an in-situ AJAX refresh of the "board" element. Notice it calls the relative URL "board". This is just a request for the active:xslt generated board fragment. My original endpoint declaration was already providing a suitable AJAX micro-channel.

Now I need to change the bindCell() function so that it actually does something when I click a cell...

```

<script type="text/javascript">
function bindCell()
{
    //alert("Binding");
    $(".cell").click(
        function()
        {
            var cell=$(this);
            $.get("move?cell="+cell.attr("id")+"&state="+player, function(data)
            {
                if(data=="OK")
                {
                    togglePlayer();
                    refreshBoard();
                }
            }
            , "text"
            );
        }
    );
}
bindCell();
</script>

```

Notice I do a JQuery `$.get()` of the relative URL "move" and provide some query parameters for the "state" (the player token value) and the "cell". This latter is a neat trick - I just find and pass the "id" attribute from the clicked cell (this happens to be the c:x:y identifier of the cell in our resource model - another example of linked data references).

I associate an anonymous callback function to the GET request. If the received data is "OK" then I `togglePlayer()` and `refreshBoard()`. So the successful move causes the board to automatically update and the player to change turns.

All I need to do now is to add a mapping for the "move" REST path. So I cut and paste the "restart" endpoint declaration and modify it to look like this...

```

<endpoint>
  <grammar>res:/demo/tictactoe/move</grammar>
  <request>
    <identifier>active:groovy</identifier>
    <argument name="operator">
      <literal type="string">
        cell=context.source("httpRequest:/param/cell")
        state=context.source("httpRequest:/param/state")
        context.sink(cell,state)
        resp=context.createResponseFrom("OK")
        resp.setExpiry(resp.EXPIRY_ALWAYS)
      </literal>
    </argument>
  </request>
</endpoint>

```

```

        </literal>
    </argument>
</request>
</endpoint>

```

Notice I source the cell and state parameters from the `httpRequest:/param/` space. Then I do a SINK request to the cell with the received state.

I refresh my browser and click a cell...

Nothing happens! I turn on the visualizer and try again. All my requests are looking good. I look at the response for the AJAX request to load the "board" micro-channel. Whoops, look at that its not rendering the *operand* correctly - I can see that it has the X value in the cell but the HTML is borked.

I look at the XSLT and realise that the xpath I used to select the cell state is missing the "cells/" part. A quick change like this does the trick...

```

<tr xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <td class="cell" id="c:0:0">
    <xsl:value-of select="row[1]/cells/cell[1]" />
  </td>
  ...
</tr>

```

I can play the game and the board starts to play like a traditional piece of paper version..

TicTacToe Demo

TicTacToe Demo

X	O	X
O	X	O
X		



[\[Restart\]](#)

Reset Shame

Having filled up a board. I hit reset. Crap. Its not resetting. The board seems to be cached. I turn on the visualizer. Everything looks ok but when I reset the cells state the active:board resource is not expired....

Now, the quick fix, is to look at the application level and to associate a golden thread with active:board and to cut the golden thread when a move is made. But that wouldn't be the normalized solution! So I dug deep.

After half an hour of digging around with the visualizer I am red-faced. I discovered that whilst the DELETE to fpds: is clearing state and cutting golden threads - there's a problem with the implementation of the trailing slash pattern in the RDBMS impl of the pds: tool.

It turns out that whilst its correctly deleting its internal golden threads - the SOURCE request for fpds: resource is only associating a golden thread for the identified resource but **not** the "trailing slash set".

So when I delete a set - any SOURCED individual items are not expired since they don't know about their membership of the set. The SOURCE needs to also associate a golden thread for the set.

Its a one line - additional argument to the call to active:attachGoldenThread in the pds impl.

With this in place. All of sudden reset works and the game is done. (I've built and shipped an update to pds-core in the apposite updates - see above).

Tidying up - a touch of constraint

So far you'll have noticed that there is no logic in our solution. There are also no constraints to prevent, for example, setting the state of a cell that has already been played. Don't worry - the constraints are still not our focus. We're still in the *composition* stage.

However, because I was publicly humiliated by the mod-pds issue - I went back to my hacky XSLT stylesheet and quickly reworked it so that it automatically renders and generates the cell "id" value. Showing that I can do things other than cut and paste(!) and also its a very small pre-cursor to the application of constraints that we'll do later...

```
<table xmlns:xsl="http://www.w3.org/1999/XSL/Transform" cellpadding="0"
cellspacing="2" class="tttboard">
  <xsl:for-each select="row">
    <tr>
      <xsl:variable name="y" select="position()-1" />
      <xsl:for-each select="cells/cell">
        <xsl:variable name="x" select="position()-1" />
        <td class="cell" id="c:{$x}:{$y}">
          <xsl:value-of select="." />
        </td>
      </xsl:for-each>
    </tr>
  </xsl:for-each>
</table>
```

Next time we'll try to make a better game than the one you can play on a piece of paper...

Checkpoint

You can download a snapshot of the www module at this point here...

- urn.org/netkernel/demo/tictactoe.www-1.1.1-r3.jar

Note: You will need to accept the update to pds-core from apposite if you don't already have it. ♠

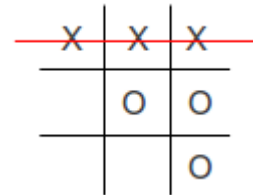
Resource Oriented Analysis and Design

Part 6

Peter Rodgers, PhD

Recap

Last week we discussed how with ROC its a useful working practice to work from both edges towards the middle. We built a web rendering layer and added the ability to make a move. By delegating state management to the edges (client-side AJAX) we implemented a first rough cut of the game.



This week we're going to add a little finesse and in the process look at stateless business logic...

Preventing replay of taken cells

Last time, we left the game in a playable condition but noted that it was violating one of the basic rules of TicTacToe in that it was possible to overwrite the state of a cell by playing another move on it. So now we need to fix this.

All that we need to do is change the implementation of the `res:/demo/tictactoe/move` endpoint so that before SINKing the move's token state to the cell, it first SOURCEs the cell and checks that it is empty (ie the default platonic empty string "" cell state)...

```
cell=context.source("httpRequest:/param/cell")
state=context.source(cell)
result=null
if(state.equals(""))
{
    state=context.source("httpRequest:/param/state")
    context.sink(cell,state)
    result="OK"
}
else
{
    result="TAKEN"
}
resp=context.createResponseFrom(result)
resp.setExpiry(resp.EXPIRY_ALWAYS)
```

The resulting representation is now either "OK" or "TAKEN". In the HTML rendering layer, you'll recall that the AJAX script only changes the client side state if "OK" is received. So a response of "TAKEN" is ignored and therefore makes a repeat move on a cell impossible.

Do we have three in a row? The CheckSet Meta Resource

To be a slightly more sophisticated solution it would be nice if we can determine if someone has won. You'll recall that back in Part 1 we anticipated this requirement in our modelling and discussed Meta Resources. In particular I proposed the idea of the "CheckSet" resource.

The idea for this resource is that for any given cell, there must be a corresponding row, column and potentially diagonal(s) that we need to look at to decide if the game has been won.

So, say we play cell `c:0:0`, then we have to check the cells in `row:0`, `column:0` and `diagonal:0` to see if any of these sets is all equal. If one is, then the game has been won.

With a code-based perspective you might start by modifying the implementation of the move service to contain the logic to source all the necessary testable cells... but that would make for a very complicated lump of spaghetti business logic. Not something you'd want to write or maintain.

The ROC way is to think of the CheckSet as a resource. There is one CheckSet for every cell on the board. You'll see what I mean if I throw you in to the deep end of my implementation. In 30 seconds I added the following endpoint in the `wwwMapperConfig.xml`...

```
<endpoint>
  <grammar>
    <active>
      <identifier>active:checkSet</identifier>
      <argument name="cell" />
    </active>
  </grammar>
</request>
<identifier>active:groovy</identifier>
<argument name="operator">
  <literal type="string">
    import org.netkernel.layer0.representation.impl.*
    cell=context.getThisRequest().getArgumentValue("cell")
    s=cell.split(":")
    x=Integer.parseInt(s[1])
    y=Integer.parseInt(s[2])
    b=new HDSBuilder()
```

```

        b.pushNode("test")
        b.addNode("id", "row:"+y)
        b.addNode("id", "column:"+x)
        if(x==y)
        {
            b.addNode("id", "diagonal:0")
        }
        if(y==(-1*x)+2)
        {
            b.addNode("id", "diagonal:1")
        }
        resp=context.createResponseFrom(b.getRoot())
    </literal>
</argument>
<argument name="cell">arg:cell</argument>
</request>
</endpoint>

```

Briefly, I've used an active grammar that expects an argument named "cell". This will be the "c:x:y" cell identity for which we require the CheckSet.

In the first 4 lines of code I extract the identity (value) of the cell argument from the request. I know the structure so I split the c:x:y string up and parse the x and y integer values. To recall the methodology of last week: Step 1 is complete. I know what I've been asked for.

There is no step 2, since I don't need any other resource to satisfy this request. In ROC terms we can think of the *CheckSet* as being a **Platonic Resource** - it is an abstract entity like a "Platonic Form", it has always been there and it will always be there (no really).

Step 3 is the *value add* part. You can see that this demands that I get off my philosophical high-horse and get down and dirty with a concrete representation. I don't waste any time worrying about the physical form of the representation, as I have explained previously, a good rule of thumb is: when you don't know or don't care then use a tree-structure.

Here, using an HDSBuilder, I construct an HDS tree and push a root node of "test". Next I add an "id" node with a value of "row:y" (the identity of the y row) and also the same for "column:x" since every cell is always a member of one row and one column.

Dealing with the diagonals requires a little consideration of linear equations. We understand that the diagonals are lines and therefore must follow the function $y=mx+c$. In our naming convention defined in part 1 we agreed that diagonal:0 goes from top-left to bottom-right so this is the line $y=x$. Equally diagonal:1 goes from bottom-left to top-right. With a moments thought its apparent this is the line $y=2-x$.

Now, we only need to consider a diagonal if our cell is actually on either of the two diag-

onal lines. So now you see what the two conditions in my code are doing to take care of this.

To see what this endpoint provides, here are three examples of the CheckSet corresponding to cells top-left (c:0:0), top-center (c:1:0) and center-center (c:1:1). As we have repeatedly seen in earlier parts, these follow the pattern of being linked data resources containing references to the identities of other resources...

Cell	CheckSet
c:0:0	<pre><test> <id>row:0</id> <id>column:0</id> <id>diagonal:0</id> </test></pre>
c:0:1	<pre><test> <id>row:0</id> <id>column:1</id> </test></pre>
c:1:1	<pre><test> <id>row:1</id> <id>column:1</id> <id>diagonal:0</id> <id>diagonal:1</id> </test></pre>

Lastly, returning to the implementation code, notice that I have not expired the response. In our constrained game board there are only 9-possible CheckSets. When we have generated one we can keep it forever. It follows that we should expect this code to run precisely nine times in the entire life of this system.

Now we need to see how to use a CheckSet...

Linked Data Logic

For each move made, the CheckSet tells us which resources need to be looked at to decide if the game has been won. So now I can enhance my move service...

```
cell=context.source("httpRequest:/param/cell")
state=context.source(cell)
```

```

result=null
if(state.equals(""))
{
    state=context.source("httpRequest:/param/state")
    context.sink(cell,state)
    req=context.createRequest("active:checkSet")
    req.addArgument("cell", cell)
    rep=context.issueRequest(req)
    ids=rep.getValues("/test/id")
    for(id in ids)
    {
        //Need to check the cells are equal
        println(id)
    }
    result="OK"
}
else
{
    result="TAKEN"
}
resp=context.createResponseFrom(result)
resp.setExpiry(resp.EXPIRY_ALWAYS)

```

After we SINK the cell state, we request **active:checkSet** for the cell.

The CheckSet's HDS tree structure allows me to select the set of id nodes from the tree with an XPath `"/test/id"`. So now I know the identity of every resource I need to look at to see if the game is over. For the moment I just iterate over them spitting out the identity of each resource.

So now we need to know when a composite resource (a set of cells) is all equal...

Set Membership Equality

I copy and paste my `active:checkSet` endpoint declaration. I delete the code and modify the grammar to `active:testCellEquality` and specify an argument name of `"id"`, the identity of the composite resource set of cells to test for equality.

Here's the implementation...

```

<endpoint>
  <grammar>
    <active>
      <identifier>active:testCellEquality</identifier>
      <argument name="id" />
    </active>
  </grammar>
</request>

```

```

<identifier>active:groovy</identifier>
<argument name="operator">
  <literal type="string">
    rep=context.source("arg:id")
    cells=rep.getValues("/cells/cell")
    result=true;
    last=cells[0];
    for(cell in cells)
    {
      if(cell.equals(last))
      {
        last=cell;
      }
      else
      {
        result=false;
        break;
      }
    }
    resp=context.createResponseFrom(result)
  </literal>
</argument>
<argument name="id">arg:id</argument>
</request>
</endpoint>

```

Using our ROC methodology again, we can analyse this endpoint.

- Step 1 (what am I promising) is explicitly declared in the grammar. I have promised that I will provide the resource that says if another resource's cells are all equal.
- Step 2 (what do I need to do this). I obviously I need the resource in question. So I must source the resource specified in the "id" argument. Notice that NKF provides a convenient convention to do this. I can SOURCE "arg:id" since the "arg:" prefix says SOURCE the resource who's identifier is the value of the argument. If the id argument is "row:0" this will SOURCE row:0. Using "arg:" is like dereferencing a pointer.
- Step 3 (add value). We know that all composite resources in the TicTacToe resource model are implemented using HDS and have the tree structure /cells/cell. So using an HDS XPath we can get the values of all the cells. The logic looks at the cells, and allowing for fast fail on a difference, tests to see if all the cells have the same value
- Step 4 (keep your promise). We return a boolean true if the cells are all equal.

Notice this code has no error handling. Why? Because if you pass it a resource reference that does not reify to an HDS representation with the correct structure it will throw an exception. We're saying "You're not keeping your side of the bargain and we're not the right re-

source for you to request - so deal with it buddy". But from the requesting side we also know that the resource identifiers that we will pass here are only ever going to be those in the CheckSet and we know from our initial construction of the TicTacToe resource model that those identifiers always reify a suitable representation.

We're now ready to finish the solution...

Final Step

So now, back in the move implementation, we can check each item in the CheckSet...

```
cell=context.source("httpRequest:/param/cell")
state=context.source(cell)
result=null
if(state.equals(""))
{
    state=context.source("httpRequest:/param/state")
    context.sink(cell,state)
    req=context.createRequest("active:checkSet")
    req.addArgument("cell", cell)
    rep=context.issueRequest(req)
    ids=rep.getValues("/test/id")
    result="OK"
    for(id in ids)
    {
        req=context.createRequest("active:testCellEquality")
        req.addArgument("id", id)
        if(context.issueRequest(req))
        {
            result="GAMEOVER"
            break;
        }
    }
}
else
{
    result="TAKEN"
}
resp=context.createResponseFrom(result)
resp.setExpiry(resp.EXPIRY_ALWAYS)
```

For each id we now request `active:testCellEquality` for that resource. (Remember the CheckSet table above, for cell `c:0:0` we request tests for `row:0`, `column:0` and `diagonal:0`). As soon as we find that a resource contains all equal cell state then the game is over. The current move is the winning move and we have a filled row, column or diagonal.

Finally, back in the HTML rendering layer, we hack the AJAX code to deal with the new possibility of a "GAMEOVER" response...

```
function bindCell()
```



```

    {
        //alert("Binding");
        $(".cell").click(
            function()
            {
                if(player!="GAMEOVER")
                {
                    var cell=$(this);
                    $.get("move?cell="+cell.attr("id")+"&state="+player,
function(data)
                    {
                        if(data=="OK")
                        {
                            togglePlayer();
                            refreshBoard();
                        }
                        if(data=="GAMEOVER")
                        {
                            refreshBoard();
                            alert("We have a winner! \n"+player+"
wins")
                            player=data;
                        }
                    }
                    , "text"
                );
            }
        );
    }
}

```

The game now plays for real and terminates naturally if a winner is found or the cells are all full.

Notes on working practice

This week I mostly refreshed and clicked cell c:o:o over and over. But AJAX is a pain to develop if you don't know a powerful trick. For the entire development and evolution of the move service I kept the visualizer on. After each change I would clear the visualizer state, click cell c:o:o and then dive in and look at the ROC domain requests.

I would do stupid things like have typos in my groovy implementation of CheckSet etc etc. The visualizer shows these exceptions and I could inspect them retrospectively at my leisure (by clicking "show response" on the red exception state).

By simple iterative refinement over the course of about 15 minutes I evolved my solution until it worked.

Next time we'll apply some final polish and add some constraints. In the mean time why not try the demo and use the visualizer to look at the requests and the cache hits. You'll find

that our resource oriented solution is minimising computational operations and has normalized the solution...

Checkpoint

You can download a snapshot of the latest modules at this point here...

- `urn.org.netkernel.demo.tictactoe.www-1.1.1-r4.jar`
- `urn.org.netkernel.demo.tictactoe-1.1.1-r5.jar`

Note: You will need to accept last week's update to pds-core from apposite if you don't already have it. ♠

Resource Oriented Analysis and Design

Part 7

Peter Rodgers, PhD

Recap

Last time we used linked data resources to determine the winning set resource. The game was essentially completed (at least as far as is necessary for the purposes of these articles).

X	X	X
	O	O
		O

Over the series so far, we have been employing an iterative development cycle moving between *Composition* and *Construction* (the first two C's).

Now we have a working solution, we can step back from the implementation details to look at the engineering integrity. Our aim today is to introduce extrinsic constraints so that in operation, the system always sits within a well defined and understood operational boundary.

We shall see that constraint should be applied with a light touch. A system with no constraint has unacceptable edges which can lead to compromised data-integrity (resource state) or operational inconsistency. However, an over-constrained system is brittle and becomes increasingly difficult to adapt and evolve.

Our ideal is a tolerant system that lies in a bounded "comfort zone" (in which data integrity is assured) but which is not internally prevented from adapting to future requirements...

Resource Model Constraint

The application of constraints necessitates us to look at the context in which our solution will operate. We can then assess the risk and introduce any necessary measures to bound the risk.

We will see that this assessment and the evaluation of the context allows us, as with the delegation of state to the edges, to seek to put our constraints on the edges of our solution as boundary conditions to the context.

So let's return our thoughts to part 2, in which we defined and implemented our atomic

and composite resources.

You'll remember that we came up with the `c:x:y` resource for a cell's state. We also reified sets of cells with the `cells{...}` resource. We showed how certain named sets of cells could provide semantic labels that would be useful to our problem - so we introduced the composite resources `row:y`, `column:x` and `diagonal:z`.

Our implementation of `c:x:y` was achieved through a mapping to the pds: (and fpds:) address space.

Our first inclination might be to put an upper bound on the size of these resource sets. This is very easy to do, we can simply limit the extent of the "platonic" reifiable resources that our address space contains (you remember how I said in part 2 we can think of the space as containing all resources?).

Modifying the grammars on our atom and composite mapper declarations can easily constrain the space. For example the unbounded endpoint...

```
<grammar>
  <simple>c:{x}:{y}</simple>
</grammar>
```

...can be "tightened up" using the simple grammar's support for an optional regex on the argument. Here is a new grammar that will only resolve `c:0:0` through to `c:2:2` inclusive...

```
<grammar>
  <simple>c:{x:[0-2]}:{y:[0-2]}</simple>
</grammar>
```

Here's a tightened up row...

```
<grammar>
  <simple>row:{y:[0-2]}</simple>
</grammar>
```

And a set of cells{} that is bounded...

```
<grammar>
  <simple>cells\{\{operand:(c:[0-2]:[0-2],?)+\}</simple>
</grammar>
```

Clearly with these constraints our resource model is now strictly bounded. Literally,

there are no other possible resources in the address space and so our model is watertight. Nothing at all can happen here that that we do not explicitly expect.

However, I would say this would be a mistake. We have assumed that the context of our TicTacToe application is universal. We have assumed that this resource model will only ever be used in that context. Why? We have constrained our model, but I think much worse, we have constrained our future opportunity.

For example, what if our TicTacToe game is so wildly popular (yeah really) and we decide we need a sequel. With our hard won market reputation for exciting cell-based entertainment, we could blow peoples minds with a [Connect-Four](#) follow-up.

You know the game, an array of 7 columns each 6 rows high. Player takes a turn to drop a Red or Yellow token into a column. It has precisely the same resource model as TicTacToe... We can get it to market in half-an-hour...

Oh wait...

Our resource model has been "welded to the TicTacToe context" - we can't do a 7x6 array with it...

I'm being deliberately ridiculous, but I'm making a serious point. An over-constrained set of resources does not provide any additional reduction in risk - but it sure does limit future opportunity. Bear with me a moment and I'll show how we can get exactly the same constraints and bounding risk for our TicTacToe and yet leave open the Connect-Four option...

For the sake of moving forward, lets reassess the cell constraint. If we simply say c:x:y must resolve any integer value of x and y then we allow for any array of cells.

So for example, our grammar would be...

```
<grammar>
  <simple>c:{x:[0-9]+}:{y:[0-9]+}</simple>
</grammar>
```

And equally we can have any number of rows, columns etc...

```
<grammar>
  <simple>row:{y:[0-9]+}</simple>
</grammar>
```

Lets assume for now that we're not going to make Chess games or Battleships, where we might want to have rows and columns distinguished by using a mixture alphabetical and integers for the x and y identity.

In the implementation you can download, you'll see I've left the resource model with these looser, but still constrained, integer grammars. But, I have to tell you I am anxious that even with this loosening, I may have over done it and closed off the *Battleships market*.

You can download the complete solution here...

- urn.org.netkernel.demo.tictactoe-1.1.1-final.jar
- urn.org.netkernel.demo.tictactoe.www-1.1.1-final.jar
- urn.test.org.netkernel.demo.tictactoe-1.1.1-final.jar

Representation Constraint

Returning to our first perspective on applying constraints - as well as tightening the resolvable set of resources, we might also start to worry about the representation state. What if you SINK something that is not an X or an O?

Just as with the grammar, at this point in our architecture, why is it of any concern to us? We can already see that Connect-Four needs the concept of a colour for the resource state.

Now what if we do decide we want to crack the Battleships market and we decide the killer feature is to offer a user editable "Battleship Designer" TM. *With our Battleships you can create your own navy...*

No, we wouldn't store bitmaps in the cells (though nothing is stopping us), we would store identifiers to the resource that is the user defined battleship. We'd use linked data even within our resource model. Why not?

So, to our general cell resource model, why on earth should we care what the representational state should be in a cell?

Boundary Conditions: Constraining the Edge

Lets move our attention up to the application itself. In the TicTacToe game, we really do care that a cell lies on the board. So we need a constraint to ensure that a move cannot be made "off-board".

We can think of the locations of all valid cells as a set. What we need is to decide is a cell

c:x:y a member of the valid set (ie is it on the board). Here's a quick and dirty way to implement this...

```
<endpoint>
  <grammar>
    <active>
      <identifier>active:validTTTCell</identifier>
      <argument name="cell" />
    </active>
  </grammar>
  <request>
    <identifier>active:groovy</identifier>
    <argument name="operator">
      <literal type="string">

        import org.netkernel.layer0.representation.impl.*
        cell=context.source("arg:cell")
        s=cell.split(":")
        x=Integer.parseInt(s[1])
        y=Integer.parseInt(s[2])
        if(!(x>=0 && x<3 && y>=0 &&
y<3))
        {
          throw new Exception("Invalid cell: "+cell)
        }
        context.createResponseFrom(cell)

      </literal>
    </argument>
    <argument name="cell">arg:cell</argument>
  </request>
</endpoint>
```

We can see that it parses the cell argument and determines if x and y lie between 0 and 2. If they do then it simply returns the cell identifier as its response. If it doesn't, it throws the teddy bear out of the pram.

So now the move endpoint can change this line...

```
cell=context.source("httpRequest:/param/cell")
```

To this...

```
req=context.createRequest("active:validTTTCell")
req.addArgument("cell", "httpRequest:/param/cell")
cell=context.issueRequest(req);
```

It will always receive a valid cell or, if an exception is thrown due to it being invalid, the move request is terminated and an ugly exception report is sent to the bad person trying to spam the game.

Here's what happens if a malicious hacker tries to play cell c:4:3...



Our TicTacToe game has a boundary condition which satisfies our initial risk evaluation. It is now guaranteed that the resource model will only contain valid cells.

Aliased Decoupling

Before we go on. Lets reconsider the implementation of the move endpoint. Notice how we changed a request for the HTTP cell parameter *HttpRequest:/param/cell* to one which asks the validation service if that resource is a member of the valid set.

But why are we hard-coding the parameter resource into the move? What if in the future the game is enhanced and we decide that the AJAX interface needs to use JSON as its resource model?

Actually, who says that the move endpoint is even going to be called by HTTP? What if we wanted to make a move by email, or using an instant message, or what about JMS (yeah really).

Why not be a little more resource oriented. A resource can have any name. Why don't we think of our address space as the abstract set and configure it so that it always has a valid resource for the current move?

Since in the last section we already provided the valid set of TicTacToe cells, we can easily provide the current move's cell identifier as a member of that set with a mapping...

```
<endpoint>
  <grammar>
    <active>
```



```

    <identifier>active:safeCellParam</identifier>
  </active>
</grammar>
<request>
  <identifier>active:validTTTCell</identifier>
  <argument name="cell">httpRequest:/param/cell</argument>
</request>
</endpoint>

```

So now we can rewrite the *move* endpoint. It becomes simpler and has no need for knowledge of the location of the HTTP parameter address space...

```
cell=context.source("active:safeCellParam")
```

Representation State Constraint

Following the same reasoning, I added a constraint on the representation state. I consider there is a set of possible valid states (in this case it has two members "X" or "O"). I then provide an endpoint that allows any resource to be tested to see if its in the set.

For good measure, I introduce the same decoupling from the HTTP parameters with an alias of `active:safeStateParam`...

```

<config>
  <endpoint>
    <grammar>
      <active>
        <identifier>active:safeStateParam</identifier>
      </active>
    </grammar>
    <request>
      <identifier>active:validTTTState</identifier>
      <argument name="state">httpRequest:/param/state</argument>
    </request>
  </endpoint>
  <endpoint>
    <grammar>
      <active>
        <identifier>active:validTTTState</identifier>
        <argument name="state" />
      </active>
    </grammar>
    <request>
      <identifier>active:groovy</identifier>
      <argument name="operator">
        <literal type="string">

```

```

        state=context.source("arg:state", String.class)
        if(!(state.equals("X") || state.equals("O")) )
        {
            throw new Exception("Invalid Cell State: "+state)
        }
        context.createResponseFrom(state)

        </literal>
    </argument>
    <argument name="state">arg:state</argument>
</request>
</endpoint>
</config>

```

So now the move endpoint looks like this...

```

cell=context.source("active:safeCellParam")
state=context.source(cell)
result=null
if(state.equals(""))
{
    state=context.source("active:safeStateParam")
    context.sink(cell,state)
    req=context.createRequest("active:checkSet")
    req.addArgument("cell", cell)
    rep=context.issueRequest(req)
    ids=rep.getValues("/test/id")
    result="OK"
    for(id in ids)
    {
        req=context.createRequest("active:testCellEquality")
        req.addArgument("id", id)
        if(context.issueRequest(req))
        {
            result="GAMEOVER"
            break;
        }
    }
}
else
{
    result="TAKEN"
}
resp=context.createResponseFrom(result)

```

And here's what a hacker sees if they try a dastardly "Battleships" move on the TicTacToe game...



Our constraints are in place. The boundary of our system is defended. Nothing that we don't want to happen can happen.

Elegance through Transreption

There's one thing that sticks out though. Did you notice that I've hard coded parsing of the cell c:x:y identifier into the `active:validTTTCell` endpoint...

```
cell=context.source("arg:cell")
s=cell.split(":")
x=Integer.parseInt(s[1])
y=Integer.parseInt(s[2])
```

If you look back to last week, I have exactly the same requirement in the implementation of the CheckSet endpoint. Clearly parsing and obtaining the x and y values from a cell identifier is a requirement of our model which we overlooked in part one.

So far we have always treated the cell identifier as a string and so we have been required to slice and dice it. What I really need is a lower-entropy form of the cell representation. I need an optimal representation that gives me rapid access to x and y.

In fact I need a representation that has an interface like this...

```
package org.netkernel.demo.tictactoe.www.rep;

public interface ICellRep
{
    public int getX();
    public int getY();
}
```

```

    public String getCellIdentifier()
}

```

and an immutable "value object" to implement this...

```

package org.netkernel.demo.tictactoe.www.rep;

public class CellRep implements ICellRep
{
    private int mX, mY;

    public CellRep(String aCellIdentifier)
    {
        String[] s=aCellIdentifier.split(":");
        mX=Integer.parseInt(s[1]);
        mY=Integer.parseInt(s[2]);
    }

    @Override
    public String getCellIdentifier()
    {
        return "c:"+mX+": "+mY;
    }

    @Override
    public int getX()
    {
        return mX;
    }

    @Override
    public int getY()
    {
        return mY;
    }
}

```

And a transreptor that, should I ask for an IRepCell, would be automatically discovered and would always give me the low-entropy representation...

```

package org.netkernel.demo.tictactoe.www.endpoint;

import org.netkernel.demo.tictactoe.www.rep.CellRep;
import org.netkernel.demo.tictactoe.www.rep.ICellRep;
import org.netkernel.layer0.nkf.INKFRequestContext;
import org.netkernel.module.standard.endpoint.StandardTransreptorImpl;

public class CellParserTransreptor extends StandardTransreptorImpl

```

```
{
    public CellParserTransreptor()
    {
        this.declareToRepresentation(ICellRep.class);
    }

    @Override
    public void onTransrept(INKFRequestContext context) throws Exception
    {
        String cellIdentifier=context.sourcePrimary(String.class);
        context.createResponseFrom(new CellRep(cellIdentifier));
    }
}
```

If I had such a thing I'd register it in my address space...

```
<transreptor>
  <class>org.netkernel.demo.tictactoe.www.endpoint.CellParserTransreptor</class>
</transreptor>
```

...and then I'd simply change **active:validTTTCell** to...

```
import org.netkernel.layer0.representation.impl.*
import org.netkernel.demo.tictactoe.www.rep.*
cell=context.source("arg:cell", ICellRep.class)
x=cell.getX()
y=cell.getY()
if(!(x>=0 && x<3 && y>=0 && y<3))
{
    throw new Exception("Invalid cell: "+cell)
}
context.createResponseFrom(cell.getCellIdentifier())
```

...oh what do you know, it was easier to do it than it was to describe.

Now that I've got this I might as well refactor CheckSet to take advantage of it too. All that's required is to source the cell as an `ICellRep`. But I also notice that I am assuming that the checkset is pass-by-reference. I need to change the move to pass it by value so that it can be transrepted (parsed). That's just a case of `addArgument()` changing to `addArgumentByValue()`...

```
req=context.createRequest("active:checkSet")
req.addArgumentByValue("cell", cell)
```

I reckon that about wraps this up. But there's just one more thing...

Context is King

Did it ever occur to you that this is a pretty crappy and unscalable solution? Did you ever think: "Yeah right - but if he's going to make his fortune he has to be able to support the playing of more than one game at a time"?

No? Well I did. Remember back in part three I said...

PDS uses the configuration resource to determine a collection name for this set of pds resources (something that is called the "zone"). You can see that in my space I provided a very simple static `<literal>` implementation of `res:/etc/pdsConfig.xml` and declared my pds zone to be `global:TicTacToe` (remember this, we'll come back to it one day).

Look at the `urn:org:netkernel:demo:tictactoe` rootspace...

```
<!--Literal PDS Config-->
<literal type="xml" uri="res:/etc/pdsConfig.xml">
  <config>
    <zone>global:TicTacToe</zone>
  </config>
</literal>
```

Now, what would happen if I moved the implementation of `res:/etc/pdsConfig.xml` from down in the low-level resource model. Up into the web application rootspace?

The answer is, absolutely nothing. PDS finds its configuration using the context of where it is requested (just as any resource is resolved in ROC).

The implementation I've used so far is an inline `<literal>`. So what would happen if I mapped `res:/etc/pdsConfig.xml` to a dynamically generated resource? What would happen if I dynamically generated the PDS config resource to be a unique zone for every unique visitor to the web page (or email listserv, or JMS message queue or ...)?

Can you see that with no change to anything other than the context of the application architecture, my application would now support unlimited, completely isolated, concurrent games? And every single isolated game would be normalized and have its state cached...



A very large representation of a TicTacToe at SunSet. ♠

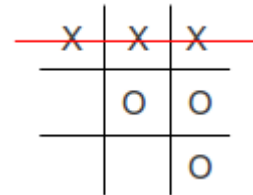
Resource Oriented Analysis and Design

Part 8

Peter Rodgers, PhD

Recap

This is the final part of this series. Now that we've implemented a solution its time to step away from the details of the problem that has been addressed and to examine the patterns, the working methodology and perhaps to extrapolate from this simple problem to bigger and more grand systems.



First though, I want to consider some feedback and questions that have come up from people following the articles...

Edge State to the Limit

Around about Part 3, Steve Cameron in Tasmania got in touch with a very good observation. He suggested that an alternative approach for the WWW application was for all of the state to reside either entirely-client side or entirely server-side. With a full representation of the state transferred with each interaction.

He has also provided an implementation of a version of the game using an XForm to show this pattern, with his comment below...

<http://collinta.com.au/xsltforms/0&X.xml>

"Its RESTful in my view as the game state (as represented by the single XForm model instance) can be easily persisted on the server by an Xform submission, and then retrieved into the form at any future point in time. Is the state in REST necessarily that of the server, I don't think so."

Here's my reply, which was sent before I started work on normalising and constraining of the solution...

You are totally correct that the state does not have to live on the server-side. In fact I was considering doing a "full game-state-transfer" version too to demonstrate a client-side state approach.

However, you are probably not my target audience for this feature. Since you have embraced a stateless and fully declarative model. This is a very long way from standard practice.

So my real intention with this series is not to show how to do a TicTacToe web application - but using this simple example to show how to step away from thinking purely in terms of code in the internals of a solution. To start to embrace declarative techniques, linked data resources etc etc.

The reason I am using server-side state - is that this constitutes the vast majority of the existing IT landscape and so is a default assumption for most developers.

What I hope I'm getting over is that state should always live on the edge (either client-side or system-of-record side) but never in the middle.

However OO design of middleware generally starts with an assumption of pulling all state into the middle.

I guess the most important thing to emphasise about this series is that it was never the intention to write a *Web-application*.

We built a web-application rendering layer just simply because it happens to be a useful and common way to expose the design requirements and to offer an implementation.

I think you can see that by the end, the services, and in particular the move endpoint have been abstracted from HTTP and could be used and exposed to other transports with different renderings (for example, you could imagine playing a 19th century style of game via sedate email correspondence, "*Dear Mr Darcy, I play an X in cell c:1:2. It is my earnest wish that you might expedite a reply forthwith. Your Humble Servant, ...*")

What about two players in different places

Earlier this week Matt Pearce at Cognizant raised another good question. How would we extrapolate last week's closing section on the migration and dynamic generation of the pds configuration state in the contextual application, so that two users could play a single game from two separate locations?

Returning back to the ROC modelling of Part 1...

With Matt's requirement we would have to introduce an additional abstract resource: "Game". We might call it `active:game+id@foo`.

As with implementing the atomic cell resources, we'd follow a similar architecture, we'd create a new module for the game resources and import this abstracted game resource model into the www tier (more on which below).

To solve Matt's requirement our pds configuration resource would therefore need to be generated with a unique zone for each Game resource.

If we were cunning about it, then we'd use the same abstraction technique as in the use of the HTTP parameter. We wouldn't hard code the pds: zone generation to request `active:game+id@foo`. Instead we would configure the space with an endpoint that provided a contextual game resource `active:currentGame`

So now, for two people to participate in the same game all that is required is that they would each share the same context for the identity of the game resource.

User Game Context

To achieve this, each player would now need to provide some state such that there was a context to enable us to determine which one was currently playing and furthermore to use this to determine the game in which they are participating.

In a web-application there are two ways to do this. The most common is to use a cookie and a correlated session (which with NK we can easily do by wrapping our application with the session overlay). The other is to have the user go through an initialisation phase where an identifying token is provided to them and stored in the client-side - in this latter case the AJAX code would need to qualify a move request with the user token too.

The `active:currentGame` resource would provide correlation between user token (or session state) and game identifier.

My inclination would be to implement this "user correlation" tier in a new module that would sit above the existing core www-module. Since the existing services are completely unaffected by Matt's new context. The user-game correlation layer would therefore play the role of contextualizing requests to the existing www game implementation.

Token State

It follows that a similar analysis and design is required to deal with the "token-correlation problem". That is, we must ensure that each person is playing the correct piece in the correct order.

Its pretty simple to sense that it would need a new set of resources

`active:userToken+user@foo` for which we'll assume the state is established during the game initialisation.

Interestingly, based on Steve's comment, this requirement can be seen as the migration of the state management of the move token from the client-side to the server-side (remem-

ber our AJAX code was previously managing this). In addition the requirement that we enforce the order in which users can play their turn also tends to favour the token state on the server-side (but not in the middle!) - though of course with a cunning token counting/coding scheme you could leave it on both client-sides.

Anyway, since we now know the user, and have a set of resources that tell us which token they must be playing, therefore we no longer need to hold that state on the client-side.

However, with this change we would be required to change last week's abstracted `active:safeStateParam` so that it would instead point to the new "contextual user token". Easy, we just change the state argument referring to the http parameter `httpRequest:/param/state` like this...

```
<config>
  <endpoint>
    <grammar>
      <active>
        <identifier>active:safeStateParam</identifier>
      </active>
    </grammar>
    <request>
      <identifier>active:validTTTState</identifier>
      <argument name="state">active:currentUserToken</argument>
    </request>
  </endpoint>
</config>
```

And of course we'd then have to abstract the call to `active:userToken+user@...` to one called `active:currentUserToken` with the user argument being resource reference to a user-correlation resource from the user-correlation layer.

And so it goes...

Process

The section above is an attempt to show how the existing game could be progressively evolved. Essentially, the process discussed is exactly the same as any of the other implementations discussed in the series...

Its the four steps.

1. What have I promised?
2. What do I need?
3. Add value

4. Respond

In the analysis undertaken to solve Matt's question I start with "I am promising a unique pds configuration for each game". I then ask "What do I need to do this". Which then leads to the revealing of the necessary services (each of which then has its own 4 step process).

You can see that to actually solve this problem for real I would introduce scaffolding resources (like the dummy resource in the early stage of developing the cell atoms). I don't have to bite off the whole problem - I can implement each resource independently and allow the interactions to emerge and be made dynamic in a progressive way.

This implementation process and more particularly Matt's new requirement are both aspects of the same thing.

We have been on a journey throughout the implementation of our solution in which we have progressively evolved it by introducing contextual layering.

This a very natural evolutionary process in ROC solutions and it comes back to something I said way back.

If you can remember back to part 2 I said imagine that an empty space contains all the resources - development is reducing the potential set down to a set that solves our problem.

The introduction of contextual layers is, just like defining endpoint grammars in a space, just another way to sculpt the set to satisfy our specification.

I was thinking this was going to be the final article, but I'm not quite done yet. Perhaps this is good point to pause. I'll offer some final thoughts next week... ♠

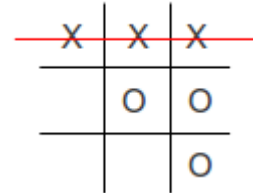
Resource Oriented Analysis and Design

Part 9

Peter Rodgers, PhD

Recap

OK, this really is the last in the series. I don't know about you but if I ever have to think about TTT again for the rest of my life it'll be too soon.



So with this in mind, lets see if we can draw out the major themes which have been woven into the details of the story so far...

Resources are Abstract, Representations are Concrete

If there is one thing to take away from this exercise it is to appreciate the distinction between a resource and a representation of a resource.

We saw through the analysis and design phase that distancing ourselves from the "baggage of representations" allows us to identify the sets of information that will inform our solution without being forced into decisions about technological delivery of the physical solution.

We don't start with code or even language. We start with identity and allow this to connect us to the concept behind the resource abstraction.

That said, we should never be "fixated by identity".

Thinking of good names is often difficult - especially when we are formulating a design. We need to be prepared to modify the names we give to things so that step-by-step we refine our system to something that can have long term stability and have identities (within the domain of expertise) that have simple and unambiguous semantics.

What I'm trying to say is - we should not be afraid to say "we don't know the answer yet".

We should be ready to make a start, knowing that it will almost certainly be revised as we learn more.

Software is always an approximation to reality - the insight I hope we're discovering is that the process of ROC software is to allow progressive and continual enhancements to the approximation - until some level of sufficiency is attained. The map maker who succeeded in making a 1:1 scale map is yet to be born.

Fundamentally ROC is a two-phase model for computation. The logical world of the resources is dealt with in the first phase - in which resolution of resource identifiers is performed. When a resource request is resolved it then enters the second phase of execution.

It follows that we can model and remodel the logical resource domain at whim and indefinitely - without having the physical domain impose itself on our imaginations.

State Transfer – Keep Transformations and State Separate

It should also be apparent that we have carefully maintained a distinct separation between representation state and transformational operations on state.

In ROC, state is transferred to code, which acts upon it producing new representations.

We should never presume that a resource will only be used in one way. We should assume a resource will be used in many new and unanticipated ways as new requirements emerge.

At first separating state and transformation (code, methods, functions) might seem quite strange - especially as it is the antithesis of the object-oriented paradigm.

But there are three overwhelming arguments why we should do this.

1. **Identity** - every resource has an identity so every computation can be cached. It is not the job of software to keep CPUs warm, its job is purely to deliver information. Nobody cares if you actually run any code at all. If we already have the desired information (whether pre-configured, cached or by transcendental meditation) then the transformation should be avoided entirely.
2. **Immutability** - extrinsic state that is transferred to the code can be accessed concurrently. The information is free - so anyone can use it anytime. The only thing that is needed is to ensure that the state is immutable. State transfer does not modify the state- it simply accesses it (hence "Accessor") and uses it to create new state. Therefore, with the ROC paradigm immutability is far easier to attain.
3. **Re-usability** - the elephant in the room of Object Oriented design is re-usability. With the best intentions we seek to create reusable solutions but honestly can you tell

me hand on heart that you actually achieve this? The reason OO is fundamentally not reusable is back to part one: "we don't know the answer yet". In fact, we never know the answer yet! The world changes all the time - software must recognise that it is "never forever". By keeping code and state separate we allow ourselves the freedom to combine the state in new unanticipated ways. We decouple the evolutionary trajectories of the logical and physical domains. In the logical resource domain we have infinite elbow room - we can introduce new resources without disturbing existing resources. Legacy resources can be used indefinitely and be combined with other resources as necessary. In ROC re-usability is a fact.

Sets Multi-valued Representations

The last section implied that if resources are sets of information then its very powerful to do collective operations on multiple-values simultaneously.

Now, there is a caveat. State transfer allows for re-usability, *provided you use a flexible data structure for your representations*.

As we saw in this series we used Tree structured data by default. It turns out that **The Tree** is a really great structure for state transfer.

To understand why, I wrote a detailed discussion, earlier this year...

ROC Data Structures

Pull not Push

You might have noticed that there were several points in the series where we created composite resources with references to other resources. So called "Linked Data" resources.

You may also have noticed that at each point where we constructed the linked data resource we actually would have had the opportunity to SOURCE the resource and combined and returned the state directly.

Essentially what's happening here is that we are imposing an architectural preference for "Pull-State-Transfer".

By which I mean - we are deferring requesting state until the absolute very last moment.

If we can get away with saying "its over there when you need it" instead of "here it is use it now" then this is a "better solution".

Why? Because in the time it takes for the state to be finally requested - someone might

have already computed it for us so we might not have to do it ourselves. Furthermore, if our use of the state has any logic involved - its possible that the logic branch we go down may not even need the state anyway - in which case we wasted our time.

Finally, by delegating to pull - we can always introduce architectural intermediary layers that provide context. Which (although it may have been hard going) we discussed last week in the concept of how to contextualize our solutions. In short, asking for it too soon, may prevent you from exploiting the power of contextual architecture.

A preference for pull should be no surprise. Look at HTML web-rendering. We don't embed the image state in the HTML page with the CSS and the scripts. The browser defers state transfer until it is requested by secondary pull requests as the HTML parser executes.

Composition Composition Composition

Putting all of the above together - it is almost a matter of inevitability that when working in ROC our time is mostly spent in composition.

I was deliberately somewhat extreme in trying to deliver an entirely compositional solution to the TTT. But its a fact that a team working on an ROC solution will spend 80% of its time in composition.

Normalization

With our presumption of "not knowing the answer yet". Its inevitable that we'll create bad and convoluted names for things. However as we start to understand the significance and conceptual value of resources then its a good thing to take the time to normalize their identity. To give things stable names with semantic integrity.

We saw that we used the mapper repeatedly to normalize and solidify the naming of our resources. Not for any technical reason but for conceptual simplicity.

Process

I've said this repeatedly but a scale invariant recipe for solving problems is to follow the four steps.

1. What is it that I'm promising?
2. What do I need?
3. Add value
4. Respond.

These steps work for an individual endpoint, a space and an entire architecture.

Was the TTT Exercise Realistic?

In many ways the TTT example was contrived and a pretty uninteresting problem. It was certainly not realistic in the sense that the state of the game was incredibly simple unlike the state of a business held in RDBMS systems-of-record.

Yet the approach and the emergent solution, the natural layering, the linked data resources, the compositional efforts, the retrospective and selective application of constraints are all 100% typical characteristics of solving real world problems with ROC.

If you've not built anything complex with NK and ROC yet then this has to be taken on trust. But here's something that might provide some scientific validation. This spreadsheet...

<https://docs.google.com/spreadsheet/ccc?key=OAJePFFtco8iCdGFHV3pwRTNpWk5mRzB6YlpYdFU2SHc>

... is an analysis of the NetKernel Portal.

The portal was developed using exactly the same approach as TTT. You'll see how many endpoints it has, how many lines of code, lines of declarative resource, use of accessors, DB tables, etc etc.

The headline is that a complete complex portal architecture was built by one person at an average rate of 45 minutes per channel (I class the entire TTT demo as equivalent to "one channel" and it took about an hour altogether - so it shows I'm getting old).

The portal has been in production for three years and has not required any bug fixes and during that time has been continually enhanced with new tools and services. ♠

Resource-Oriented Computing and Programming Languages

- Introduction
- Imports and Extrinsic Functions
- Arguments
- What is Code? What is a Function?
- Functions, Sets and RoC
- RoC: State of the Union

An illustration of [Cantor's](#) diagonal argument for the existence of uncountable sets. The sequence at the bottom cannot occur anywhere in the list of sequences above.

The diagonal argument demonstrates a powerful and general technique that has since been used in a wide range of proofs, also known as diagonal arguments by analogy with the argument used in this proof. The most famous examples are perhaps [Russell's paradox](#), the [first of Gödel's incompleteness theorems](#), and [Turing's](#) answer to the [Entscheidungsproblem](#).

Source: Wikipedia

$E_0 =$	m	m	m	m	m	m	m	m	m	m	m	m	...
$E_1 =$	w	w	w	w	w	w	w	w	w	w	w	w	...
$E_2 =$	m	w	m	w	m	w	m	w	m	w	m	w	...
$E_3 =$	w	m	w	m	w	m	w	m	w	m	w	m	...
$E_4 =$	w	m	m	w	w	m	m	w	m	w	m	w	...
$E_5 =$	m	w	m	w	w	m	w	m	w	m	w	m	...
$E_6 =$	m	w	m	w	w	m	w	m	w	m	w	m	...
$E_7 =$	w	m	m	w	m	w	m	w	m	w	m	w	...
$E_8 =$	m	m	w	m	w	m	w	m	w	m	w	m	...
$E_9 =$	w	m	w	m	m	w	w	m	w	w	m	w	...
$E_{10} =$	w	w	m	w	m	w	m	w	m	m	w	m	...
$E_{11} =$	m	w	m	w	w	m	w	m	m	w	m	m	...
\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	...
$E_u \neq$	w	m	w	w	m	w	m	m	m	m	m	w	...

Resource-Oriented Computing and Programming Languages

1 – Introduction

Peter Rodgers, PhD

Its about ten years since we created the first ROC language runtime resolved in a Resource Oriented Computing address space. For the record it was `active:xslt`.

You maybe don't get off on writing your code in XSLT but it is Turing complete - so it is fundamentally an equal with Groovy, Python, Javascript, Java, Clojure, Scala, Ruby, Bean-shell or any of the other languages that you can program in on NetKernel today.

Here's something else to make you think. Apart from natural evolution of the black-box internal implementation, the XSLT runtime's logical ROC interface remains the same today as it was then; therefore given the same xml document "`myDoc.xml`" and the same transform "`myTransform.xsl`", then an ROC request for:

```
active:xslt+operator@myTransform.xsl+operand@myDoc.xml
```

gives you exactly the same xml resource today as it did then.

So the set of accessible resources computable with the XSLT runtime (and by Turing equivalence any other language) is very stable, even across generations of the NetKernel ROC system.

Its time to explore why ROC can make promises that deliver this kind of stability and, ultimately, examine how ROC sheds light on the nature of how we think about code, languages and computation.

Over the course of the next couple of newsletters I want to:

- Explain the role and concept of languages in ROC.
- Show how "Code State Transfer" shows the language runtime pattern to be just a special case of an arbitrary resource accessor.
- Describe the role of Turing complete language runtimes

- Explore why Turing equivalence is not necessary for an ROC system.
- Consider how ROC's extrinsic context can influence language design.
- Discuss the contextual trust boundary constraints required for "open computation" in an ROC space.

I hope that I'll be able to convince you that languages are second order. That as software engineers we can "step out of the 2D plane" of languages, programming and code. From a new "3D perspective", see that code is a means to an end, not the end itself.

Ultimately I hope to persuade you to look at the world as sets of information resources. Like Michelangelo releasing David from the marble block, information engineering is the art of applying context to release information.



I don't know if I'll be able to pull this off. But here goes...

Code State Transfer - Language Runtime Pattern

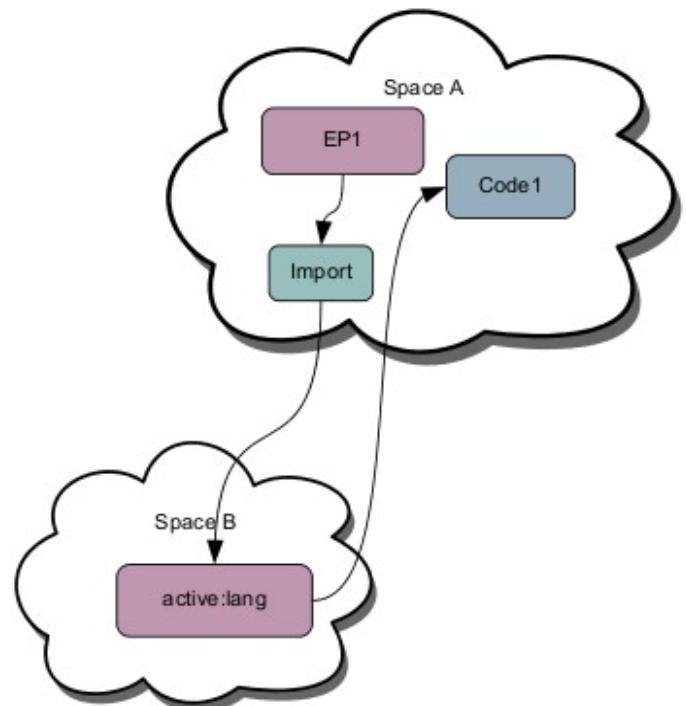
Firstly you have to forgive me. We have been living with ROC and language runtimes for a long time and we tend to take them for granted. Hence the ROC Basics tutorial I wrote...

<http://docs.netkernel.org/book/view/book:tutorial:basics/doc:tutorial:basics:part5>

where I pragmatically introduce language runtimes as script execution engines and just assume you'll "get a feel for it". Fortunately people do, but there's a lot of implicit knowledge assumed.

So let's go back to first principles and examine how the language runtime pattern works...

Let's consider an Endpoint **EP1** that wishes to execute some code **Code1** - for the sake of this example let's assume EP1 and Code1 are in the same address space **SpaceA**. Let's assume there is a library address space **SpaceB** that provides a service **active:lang** which is able to execute



Code1; and SpaceB is imported into SpaceA via an import endpoint. The diagram below shows the spacial architecture...

Lets assume that **active:lang** uses the conventional active URI (Butterfield, Perry, IETF Draft 2004) syntax for a language and supports an operator argument specifying the code (script) to execute.

Therefore to execute Code1, EP1 would issue a request...

active:lang+operator@Code1

This initiates the following sequence...

1. The request is resolved via the import to **SpaceB**.
2. **SpaceB** resolves the request to **active:lang** endpoint.
3. The **active:lang** endpoint receives the request for evaluation
4. **active:lang** issues a request for the operator argument, in this case **Code1**
5. The request for **Code1** is not resolved in **SpaceB** and is delegated back to the next nearest request scope **SpaceA** (we sometimes call this resolving "up the superstack", or in "the request scope").
6. **SpaceA** resolves the request to the endpoint providing **Code1** (we've not stated the nature of this endpoint, it might be a fileset, or could be dynamically generated. active:lang doesn't care).
7. The **Code1** representation is returned to **active:lang** whereupon it is evaluated.
8. When **Code1** has completed, **active:lang** returns the response computed by Code1.
9. **EP1** receives the computed representation resulting from the execution of Code1.

At face value this seems like a lot of work. But there's even more! To show the broad steps I deliberately missed out some of the background detail...

Compilation Transreption

Let's assume that SpaceA's Code1 endpoint provides a binary stream representation (not uncommon for code coming from a module on your file system). Let's also assume that active:lang is implemented to run against the JVM and the implementation executes JVM bytecode.

- When **active:lang** issues its request for the operator **Code1** it indicates that it re-

quires a `LangByteCode` representation (in spaceB lets assume there is a representation class called `LangByteCode` - a value object holding byte code of the given lang).

- When `Code1` is returned to **active:lang** the kernel determines that `Code1` is a binary stream.
- The kernel issues a `TRANSREPT` request to resolve an endpoint that can transrept binary streams to `LangByteCode`.
- SpaceB has a transreptor that compiles this language (quite naturally a runtime and its compiler transreptor are usually located in the same library address space)
- The compiler compiles (transrepts) `Code1` to `LangByteCode`.
- The `LangByteCode` is returned to the language runtime for evaluation.

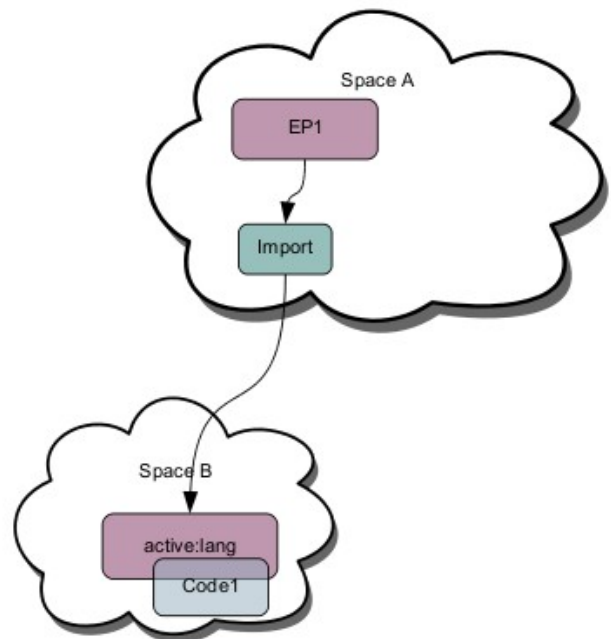
That's the whole story. Wow a lot goes on. Why the heck would you go to all this trouble just for `EP1` to run `Code1`? Surely I'd be better off having a function inside `EP1` that does the same job?

Return on Investment

Ah but that's to overlook even more detail... As the chain of requests were issued the relationships between the spaces, the endpoints and the resources were fully determined and all the state was cached.

So what happens the next time `EP1` wants to execute `Code1`?

1. The request is immediately issued to `active:lang` for evaluation (The resolution doesn't happen - it was cached in the resolution cache)
2. `active:lang` requests `Code1` which is instantly returned as `LangByteCode` from the representation cache.
3. `active:lang` executes `Code1` and returns the representation (more on that later).



Just as a HotSpot Java VirtualMachine JIT compiles bytecode to native code; NetKernel is "JIT compiling" both the code *and* the spacial context in which the evaluation of the code is occurring.

Because NetKernel/ROC discovers and accumulates optimized state, the effective run-time state of the system is to conceptually consider that the Code1 resource has been "state transferred" with minimum entropy to "active:lang" as shown in this diagram...

The computational cost of EP1 going extrinsically through the ROC address space to execute Code1 is, as near as damn it, no different to if it had a local hard-coded internal function (in real terms, the "abstraction cost" is two constant time map lookups).

The benefits are significant, just in physical terms (not dynamic state minimization - we'll cover that later).

- EP1 and Code1 are "dialectically decoupled" - they do not need to be written in the same language.
- Execution of Code1 automatically scales linearly with CPUs.
- Code1 can be executed asynchronously just by issuing the request with `issueRequestAsync()`
- EP1 can be "logically decoupled" from Code1 by simply introducing a mapping in SpaceA to, for example, have an alias like this...

```
myService -> active:lang+operator@Code1
```

- If we decide that Code1 is a critical performance hotspot and needs to be optimized and written in Java then it can be replaced with a Java accessor class with no change to EP1 (which may *not* be written in Java itself).

Where is the Code? I don't care, it's a resource

Now in this example we co-located Code1 with EP1 since it allows us to show a typical pattern for contextual application spaces. But Code1 is just a resource. What if I set up my special context so that Code1 was mapped to an `http://` URI and we imported `http-client` library. Or Code1 was a `data:` URI - the identifier and representation are the same?

Equally we could have an endpoint co-located with the language runtime, a stateful address space to which we could SINK and SOURCE code representations. EP1 can request the execution of code located in the remote address space.

With ROC we have complete architectural freedom to partition state where it is computationally cheapest - this state partitioning invariably is a runtime consideration and has no bearing on the logical composition of a solution.

Which is why long-term system stability is invariant with respect to physical and architectural change. Or, to put it another way, it should be no surprise that 10-year old systems keep on running across generations of physical platform.

Summary

This has been a fairly dry step-by-step description of the language runtime pattern. In practice you never think about this - you just issue (or map) a request to your preferred language runtime and tell it what code to run. The detail is irrelevant to a solution developer.

Next time I'll talk about the relationship between function arguments, resources, languages and execution state. ♠

Resource-Oriented Computing and Programming Languages

2 – Imports and Extrinsic Functions

Peter Rodgers, PhD

In the last gripping episode we saw how the transfer of code as state to a language runtime (**code state transfer**) introduces flexibility and liberates a system architect to step-outside of language and choose the dialect of code that most readily solves the problem.

In this instalment I want to show how the language runtime pattern changes the way we can think about *imports*.

Import

I'm willing to bet that every language you've ever coded in has an import statement of one kind or another.

So when you declare an import within a language what are you doing? You're indicating to the compiler, or interpreter, that within its accessible resources it should load the specified code and make the functions and/or objects referenceable to the code that follows the import.

Imports: A brief survey

Different languages have different models for what we mean by accessible resources. For example many Unix heritage languages, like Python and Ruby, have the concept of a library **PATH**; a set of directories within which an import reference will be attempted to be resolved and matched to a corresponding piece of code.

Java, coming out of Sun and having something of a Unix heritage, has a similar PATH model but it is a slightly more general implementation. By default the file system is abstracted into the **CLASSPATH** which comprises a set of directories or jar files from which compiled java classes may be resolved. But Java goes beyond this and allows you to introduce your own *ClassLoaders* such that imports can be resolved by arbitrary physical mechanisms. (NetKernel implements its own classloaders to provide a uniform and consistent relationship between the ROC address space and physical modular class libraries).

Languages like *XSLT* are even more general, all imports are referenced by URI and are delegated to a *URIResolver* - the language doesn't actually specify how or what this should

do, although given its W3C heritage the inference is that it should issue "GET" requests (or their equivalent for protocols other than http: such as file: URIs).

Javascript, being a language originating within the context of the Web actually has two models, it is assumed that the execution context is extrinsic (the browser) and that extrinsic context will provide a global namespace of functions aggregated from all the script references in the HTML page. Javascript is security conscious and has no native internal import, but it does offer an indirect equivalent: you can `eval()` a script resource requested via AJAX.

Irrespective of which of these you're familiar with, one more observation, *the imported code resource is always in the same language*.

Language Runtime Imports

Now every language runtime that comes with NetKernel has the ability to import. Usually we've been able to abstract the language's internal mechanism away from the innate hard-coded assumptions of that language and **ROCify** it.

For example both `active:python` and `active:ruby` support a PATH model, but the path is a set of logical ROC addresses decoupled from a physical filesystem. Both languages have companion language libraries provided as an address space in a module. With their resource loading abstracted to use the ROC domain they have absolutely no idea that *NetKernel's not Unix* (sounds dangerously like GNU eh?).

The `active:xslt` runtime has a *URIResolver*, its just that its resolver is *NetKernel*, so xslt code has unlimited scope to work with resources in the ROC domain.

NetKernel's server-side `active:javascript` runtime supports imports of the Java class path. There's also a special global object called `contextloader` which is like the XSLT URI resolver and will import scripts as ROC resources. For good measure you can also use the `context` object to source and `eval()` scripts.

There's an immediate benefit to all of NetKernel's language runtimes in having their imports come through the ROC domain. They all acquire the innate dynamic reconfigurability enabled by NetKernel's modularity. Also, wherever possible, they are implemented with compiling transreptors so that the imported code is JIT compiled.

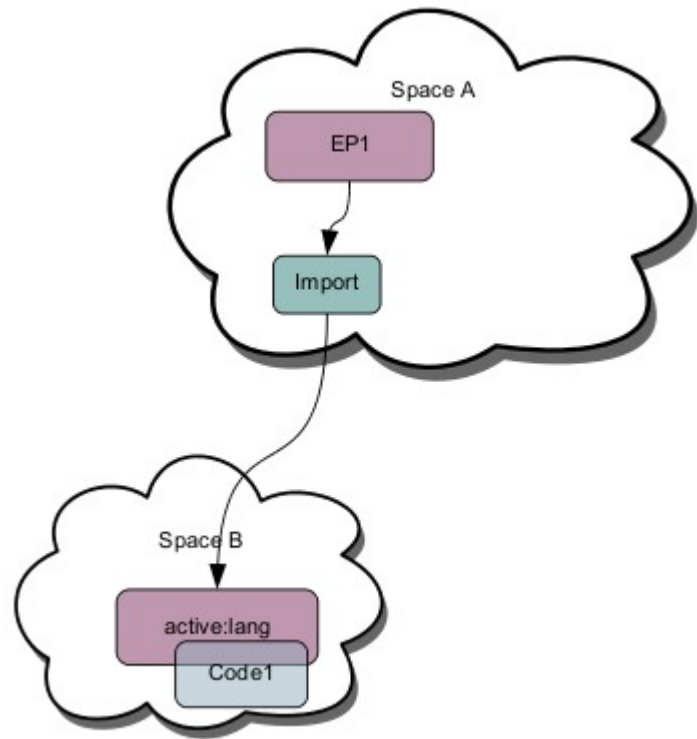
So being a language in NetKernel's ROC address space is a friendly and flexible place to be. But there's more...

Why are you worried about imports? Is it maybe because you're still thinking inside the language?

Import or Extrinsic Functions?

Remember our example from last week, we showed how the language runtime pattern allows an Endpoint to execute arbitrary code in an arbitrary language and with NK's ability to accumulate transient representational state, the net cost of the pattern is negligible compared with a local function implementation.

We closed last time with the following diagram and characterized this dynamic state as "code state transfer" to the runtime endpoint...



Now what if Code1 itself embodies another request with a reference for another resource? (As we've seen this could be implicit via an import statement - but let's not dwell on that case). It doesn't matter what the details of the actual request are, for example it might be...

```

var localState="CUST:12345678"
req=context.createRequest("active:balance")
req.addArgument("customer", localState)
var balance=context.issueRequest(req)
  
```

Which is NKF pseudo code to issue a request for the resource

```
active:balance+customer@CUST:12345678
```

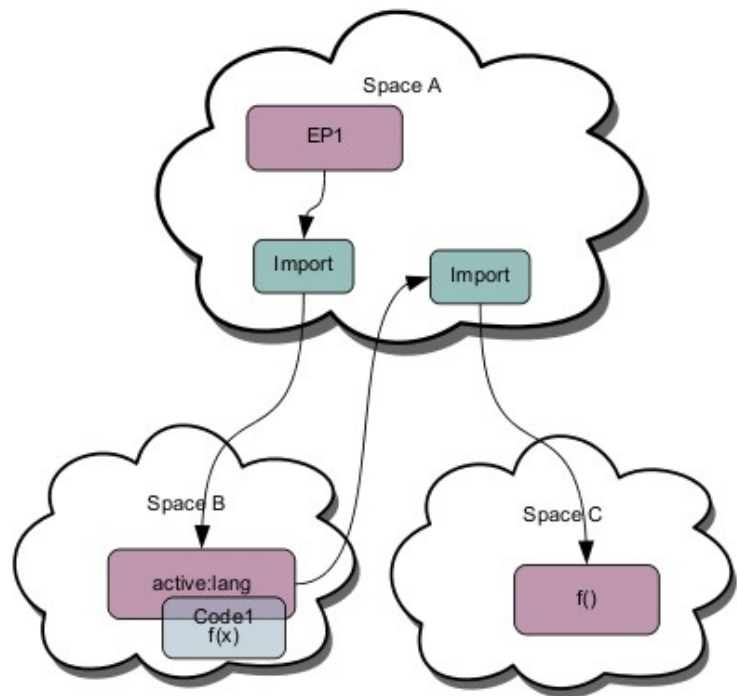
For our discussion, it doesn't matter, we can consider this as the invocation of an extrinsic function $f(x)$ where x is some internal state within the execution of Code1.

Does the language know anything about $f(x)$? No this function is outside of the current language - all Code1 knows is $f(x)$'s identity (we'll see later that it doesn't even need that).

So does SpaceB (the home of the language runtime and the immediate scope for the execution of Code1) import f(x)? No SpaceB doesn't know anything about f(x) either.

So does SpaceA (the home of Endpoint1 and the endpoint that originates Code1) implement f(x)? Not necessarily!

What if f(x) is implemented in **SpaceC** and **SpaceA** has an import endpoint to SpaceC. In this configuration here's what happens when Code1 initiates a request for f(x)...



So you can see that the concept of an "import" in ROC is extrinsic and is defined by the architectural arrangement of spaces.

Where's the function? I don't care, I'm not running code, I'm asking for a resource.

What if (as we showed in last weeks discussion of active:lang requesting *Code1*) *SpaceB* doesn't import *SpaceC*, but maps f(x) to a remote request, say to an http resource...

f(x) -> <https://bigbank.com/mybalance/x>

(presumably with all necessary credentials, perhaps using *OAuth!*)

If we do this mapping does *Code1* or the *active:lang* language runtime need to import an http-client or *OAuth* support? Nope, they're extrinsic, provided by the calling context.

Finally, just as last week's discussion showed that the execution of Code1 is, near as damn it, equivalent to a locally coded function, so the same argument holds for the extrinsic evaluation of f(x). Its as cheap to request f(x) extrinsically as it would have been to implement f(x) inside Code1, or to have used a classical language import of a library containing f(x).

Of course you also know without me saying, f(x) doesn't have to be written in the same

language as Code1.

So with the extrinsic evaluation of functions offered by ROC, we are even more liberated from language. We are also liberated from execution on the current physical hardware platform (which we indirectly discussed here).

From Javascript to Extrinsic Contextual Functions

Hopefully you'll be seeing that invoking a resource request from within an arbitrary piece of code, is a generalization of how Javascript taps the extrinsic context of the browser HTML page evaluation. Only unlike the Javascript/Browser analogy, NetKernel is providing a normalized language neutral execution context and that context is multi-dimensional.

Extrinsic Recursion

For the computer scientists amongst you, consider that $f(x)$ can be a recursive request - for example it could be...

```
active:lang+operator@Code1+x@foo
```

Recursion lies at the heart of computability. Later on we'll show that when you make recursion extrinsic you step out of the plain of language and some really interesting things happen. (If you can't wait here's a taster).

Status Check

The transfer of code as state to a language runtime ("code state transfer") introduces flexibility and liberates a system architect to step-outside of language and choose the dialect for code that most readily solves the problem. We saw in the last section that it also liberates a language from being bound to its functions. "Import" need have no relation to the chosen language, it is an extrinsic concept defined by spacial context.

Finally some practical thoughts on what we've learnt so far...

Being freed from language gives you freedom of choice. Which language you choose might be motivated by technical considerations: a given language has a particular capability (like Javascript can natively work with JSON). It might be practical, for example declaratively processing XML with XSLT can often be trivial compared with using APIs and document object models. To risk going all Zuckerberg, it might even be motivated by a social constraint: your developer knows and is comfortable with language X.

Standing back, with Turing equivalence and dynamic compiled transreption to byte-code, the choice of language is a secondary consideration.

Interface Normalization

You know how, when you go to a large web property and you see a URL like `http://bigbank.com/youraccount.asp` or for that matter `http://bigbank.com/youraccount.jsp` - you feel a sense of sadness? No, it's not the negative balance of the account that's upsetting, it's seeing reasonable and sane engineers enshrining the technological implementation into their public resource identifiers.

Don't switch off, this ain't your usual REST zealotry.

Frankly, I understand that the platforms they've chosen actually pretty much require that the extension be used to bind to the execution. I find it deeply frustrating that this also just happens to play nicely to a vendors long-term lock-in of the customer. But alas even with the high intellectual level of software purchasers, time and again the IT industry finds ways to persuade customers to neglect to follow *caveat emptor*

Well even though ROC allows you to step out of language, be aware that as an architect you need to be diligent and be aware of the pitfalls of locking in your interface to your implementation. Here's some simple rules of thumb

1. A local set of services can invoke one another by calling the runtime and expressing the code
2. If those services are exposed publicly, for example as a library, you should normalize the interface and hide the code.

For example, `active:lang+operator@Code1` is fine if that's local functionality but if `Code1` is actually generating `youraccount` balance then please implement a normalized interface mapping something like this...

```
<mapper>
  <config>
    <endpoint>
      <grammar>active:accountBalance</grammar>
    </endpoint>
    <request>
      <identifier>active:lang</identifier>
      <argument name="operator">Code1</argument>
    </request>
  </config>
</space>
```

```
...Code1 and active:lang are in here...  
</space>  
</mapper>
```

I've gone on long enough this week. I was hoping to cover the relationships between languages, functions and arguments, but that'll be for next time. ♠

Resource-Oriented Computing and Programming Languages

3 – Arguments

Peter Rodgers, PhD

In the first article in this series I covered the basics of the ROC language runtime pattern and discussed how, by treating code as a resource, we get something we've called code state transfer.

In the second article I discussed the general nature of imports and surveyed a variety of languages detailing some typical models for accessing code resources referenced via internal import statements.

We saw that languages running in the ROC domain can have internal imports that, like XSLT's URResolver, can transparently exploit the external ROC resource space and thereby gain dynamic modularity.

But further, we saw that thinking "outside the box" of the language we can make requests from within our code out to extrinsic functions. We discovered that with extrinsic functions the concept of an "import" also becomes extrinsic and is a property of the spacial architecture.

We found that by extending our view from thinking in terms of running code, to requesting normalized functions, we gain the ultimate in loose coupling - allowing the implementation of the function to be abstracted, to be scaled out across local physical computation resources and even to be externalized into the cloud.

Arguments

ROC allows us to adopt an extrinsic view of functions. But a function must be told what it is to operate upon. For that, a function needs arguments. As you'd expect given the trend we've established, arguments in ROC are also extrinsic...

Lets take these last few sentences at face value. Further along with the story we can return to this and explore the implicit underpinnings in more depth. For the moment, we'll examine the specifics of how NetKernel allows arguments to be supplied to a function.

First off, what follows is a discussion of how an NK/ROC endpoint embodying some function may receive arguments. We have stepped out of the plane of a specific language. In particular we will consider pass-by-reference and pass-by-value and these are specifically

constrained to the ROC meaning.

As you know, NetKernel is implemented on the Java virtual machine, but in what follows we are not concerned with the specifics of the Java language. (For a quick recap of the Java model see this article.)

One more bit of preamble. We'll use pseudo code examples using the NKF API however everything discussed also holds for the declarative request and where necessary we'll illustrate the equivalent construction. This is an important point, since it means that the ideas discussed here are equally applicable in any of the higher-order declarative technologies of NK such as the mapper, XRL, DPML etc.

Scenario Recap

Recalling the configuration we've used in the earlier articles. Lets assume we have a requesting endpoint **EP1**. This endpoint wishes to invoke a language runtime `active:lang` with some code **Code1**. Lets also assume that we are going to supply an additional argument which will be used by the code. Lets call this argument **operand**.

So the active URI structure of EP1's requests will look something like this...
active:lang + operator@Code1 + operand@ [...to be discussed below.....]

Arguments Have Names

The first observation is that NK arguments are named. This is different to Java where they are positional.

In our scenario we have two named arguments: operator and operand

The order in which you specify named arguments is not significant to the requested endpoint. However, the NKF core will always normalize a constructed resource identifier by sorting the arguments by name.

For example to the `active:lang` endpoint these two requests are equal...

```
active:lang + operator@Code1 + operand@xxx
active:lang + operand@xxx + operator@Code1
```

However, when you programmatically construct a request (NKF or declarative) it will always be sorted before it is issued. So, in the visualizer for example, you'd only ever see the second one (with operand ahead of operator).

By sorting by argument the infrastructure is taking care of you to ensure that you always have normalized identifiers which a) minimizes the representation cache size, or vice-versa,

b) maximizes your cache hit rate.

Again, to make progress we will take named arguments as a given. We may get around to showing later that this is not a hard coded constraint but purely a convention of the higher-order Standard Module. To be absolutely ROC purist, an argument is just a unique subset of an opaque resource identifier token and, in fact arguments are a higher-order construct. The NetKernel kernel has no concept of arguments!

Pass By Reference

Lets imagine that EP1 knows the identifier of some resource **res:/X**. EP1 wants Code1 to process that resource. It could construct and issue a request like this...

```
//Inside EP1
req=context.createRequest("active:lang")
req.addArgument("operator", "Code1")
req.addArgument("operand", "res:/X")

result=context.issueRequest(req)
```

Which would construct and issue the following **SOURCE** request (notice that we've accepted the default SOURCE verb for the createRequest(). Also note the sorting)...

SOURCE: active:lang + operand@res:/X + operator@Code1

You can see that the request construction has added the named arguments and each argument is a resource identifier. Therefore these arguments are pass-by-reference.

Evaluation

The request is resolved to the **active:lang** endpoint which sources and executes Code1 (as discussed at length in the first article).

So now lets assume that Code1 is written to perform some work for us on the operand argument. Code1 can obtain a representation of the **operand** argument by making a **SOURCE** request for it.

One way it can do this is to inspect the invoking request and to obtain from it the value of the named operand argument...

```
//Inside Code1
receivedRequest=context.getThisRequest()

operandIdentifier=receivedRequest.getArgumentValue("operand")
```

```
//At this point: operandIdentifier == "res:/X"

operandRepresentation=context.source(operandIdentifier)
```

When this code is executed in Code1, it SOURCES a representation of the state of res:/X and can do whatever it is that Code1 does.

The constraint is that, just like a pointer in C or a reference in Java, the representation held by Code1 is not intrinsic to it. It is extrinsic and any number of other threads could also be accessing it.

Therefore you must take care to treat the representation as an immutable value object. Immutability ensures safe and scalable concurrency and is now regarded as a requirement for multi-core concurrency in the new generation of languages such as Clojure etc.

Representation Immutability is a strong rule. However there are circumstances where you as an architect are completely sure of the extrinsic context and can bend the immutability rule - but you are taking responsibility - if you change a representation value you have stepped beyond the range of NetKernel's safety net.

Dereferencing

By explicitly inspecting the request, the invoked endpoint gets comprehensive access to everything that is known about the incoming request. However this is generally not something that concerns us.

Mostly all we really care about is *dereferencing* the argument reference to get the representation.

The NKF API and declarative requests, allow us to do this by using the local **arg:** identifier scheme. This is easiest to see with an example. Here's some code that does exactly the same as in the previous explicit form...

```
//Inside Code1
operandRepresentation=context.source("arg:operand")
```

SOURCEing *arg:operand* dereferences the "operand" pass-by-reference argument and all of the baggage goes away.

Transreption: Type Decoupling

Of course as soon as we drop down inside the execution of the language we've chosen for Code1 we are intrinsically constrained to a given API. It follows that Code1's use of the the

received argument will also conform to a given internally chosen API. Otherwise we'd have a *type mismatch*.

So, are we saying the origin endpoint for *res:/X* has to share the same type model as *Code1*? If so, then our journey to explore extrinsic decoupling becomes a lot more tightly constrained!

Fortunately, that's not the case...

EP1 doesn't need to care what the representational form (type) of *res:/X* is. The end-point providing *res:/X* doesn't need to care what language or representational resource model someone requesting *res:/X* might use. And to complete the triangle, *Code1* can be coded to its own internal representational model too.

Alchemy? Well yes, kind of. In ROC we call it **transreption**.

Let's imagine a concrete example, suppose that *res:/X* is a binary stream (maybe its really coming from a remote HTTP GET request or a file resource. Within *Code1* we don't know about the mapping - its extrinsic!).

Here's all that *Code1* needs to do if it wants to work with that resource as a String...

```
//Inside Code1
operandRepresentation=context.source("arg:operand", String.class)
```

We are explicitly stating that we require the argument "operand" as a String. We're also introducing an extrinsic type constraint. We're saying, if the extrinsic context can't get us the argument as a String then the architect needs to sort it out!

Usually an architect sorts it out by importing a resource model library that provides transreptors that impedance match representational types that are expected to be used. So SpaceA where EP1 lives, would probably import layer1 which has a bunch of basic type Transreptors for convenience.

Type conversion is *extrinsic* and not a concern of *Code1*. Nor is it a concern of the *language runtime* or its library Space either.

So we now find that extrinsic functions are logically decoupled and **type decoupled**. *Looooooooooooooooose coupling!*

Pass By Value

Often you will have a situation where you have a local value object in the execution state of

your code and it is this *intrinsic* state that you want to process with an *extrinsic* function.

For example suppose we have some *Stuff* in **EP1** and we want to have Code1 process it? Here's how we'd do it...

```
//Inside EP1
localState=new Stuff();

req=context.createRequest("active:lang")
req.addArgument("operator", "Code1")
req.addArgumentByValue("operand",localState)

result=context.issueRequest(req)
```

Notice that we now explicitly indicate that the `localState` object is to be passed-by-value by using the **addArgumentByValue()** construct. The named **operand** argument is now pass-by-value.

If you're familiar with REST, you might be thinking this is analogous to a POST - there is some local state and we require it to be *PUSH-State-Transferred* to the remote endpoint.

Notice that we've not changed the verb, we're still constructing a SOURCE request. Surely this is a violation of all that is good and true. We are spitting in the face of Resource Orientation. We are defiling the holy temple... We are....

Not so fast REST-boy. Stand-back this calls for ROC-man...

If, like a POST, we were to place the pass-by-value state representation inside the request then, just like a POST, that state would leave the address space and become dark-state.

As any good REST-afarien can tell you, you know what happens with dark-state don't you? The representation that is computed by the extrinsic function is no longer unambiguous, it cannot be given to the next person that asks for it since we don't know what was in the hidden layer.

Writ large on the tablets of stone in every REST tutorial will be: **You cannot cache a POST.**

Bzzzzzzzzzzzzzzzzzzzz. Wrong.

Or at least right, if like the Web, you only have a *partially resource oriented abstraction*. With ROC we have a *general resource oriented abstraction* of which REST and the

Web are a subset - this changes everything.

In ROC we have **space** as a degree of freedom and, as we have seen throughout this series, we have **extrinsic contextual relativity**.

Let's see how this gives us a *Get Out of Jail* free card for pass-by-value arguments...

Space and Pull-State-Transfer

When you `addArgumentByValue()` *NetKernel* doesn't shove the value object inside the request like a *POST* body.

It uses ROC's new dimension.

It constructs a transient space and places the value-object as a resource in that **pass-by-value space**. It gives the resource an identifier too, a made-up unique identity, typically with the scheme **pbv:**.

The request construction is exactly the same as in the pass-by-reference, only now the *pass-by-value argument* has the resource identifier to the resource in the **pbv:** space.

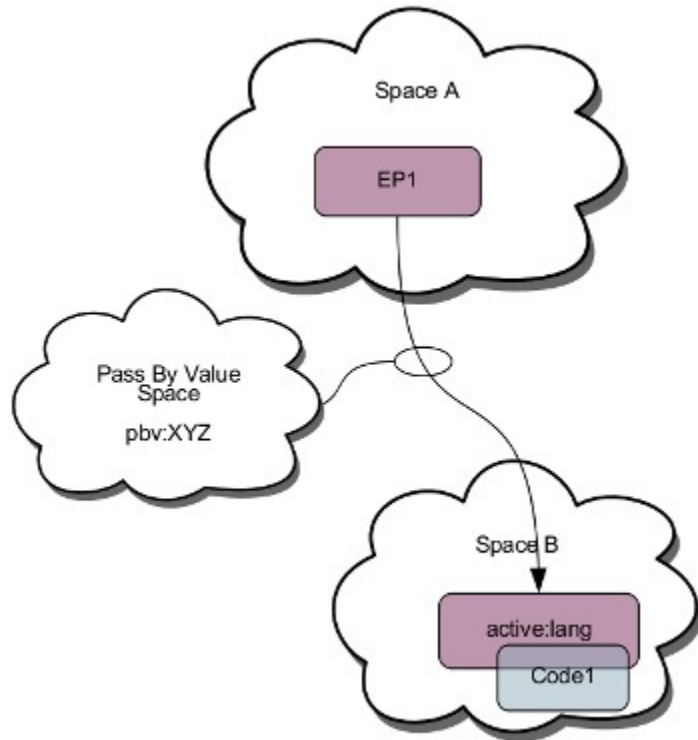
When the request is issued, *NetKernel* injects the pass-by-value space into the request scope. You can see the spacial structure in the diagram on the left.

This code ...

```
//Inside EP1
localState=new Stuff();

req=context.createRequest("active:lang")
req.addArgument("operator", "Code1")
req.addArgumentByValue("operand",localState)

result=context.issueRequest(req)
```



...therefore causes the following SOURCE request to be issued...

SOURCE: **active:lang** + **operand**@pbv:XYZ + **operator**@Code1

Where, lets assume the localState object was placed into a PBV-space under the resource identifier **pbv:XYZ**.

Requested Endpoint

So what are the consequences for the requested endpoint? Answer: absolutely nothing, the endpoint has no interest in the nature of *by-reference* or *by-value* arguments.

With PBV-arguments the examples we showed of Code1 obtaining the operand argument are exactly the same. Just for detail, here's the first example if it was requested with the *pbv:XYZ* argument ...

```
//Inside Code1
receivedRequest=context.getThisRequest()

operandIdentifier=receivedRequest.getArgumentValue("operand")
//At this point: operandIdentifier == "pbv:XYZ"

operandRepresentation=context.source(operandIdentifier) //Resolved in the PBV-space
```

PBV Consequences

It should be clear that, to the ROC abstraction, pass-by-reference and pass-by-value are the same!

It follows that all the other properties hold true also. Transreption of the PBV state occurs automatically and as required by the needs of the requested endpoint. **Type decoupling holds.**

Furthermore, since the PBV is a true resource in a true resource space. Any resource computed based upon its value is cacheable. **The extrinsic memoization of functions holds.**

It may be that the PBV state has limited commonality for future requests. For example EP1 might compute new localState for every request it receives. In which case the result of EP1 will have no cacheable value.

However what if in the request below EP1, *Code1* is a long lived process, making many

requests, all of which rely on the state of the `localState` received from EP1? All of these *micro-processes* generate micro-resources all of which can be cached since the *localState* is static in the PBV space and *extrinsic to them*!

It might come as a surprise but there are many many real world problems where outer (apparently transient) state can be made extrinsic and allow for micro-caching and dramatic computational efficiency. What about dealing with external HTTP POST requests? NK/ROC will internally cache the intermediate state while we're processing them. You can cache inside a POST!

Pass By Request

There's one more trick up ROC's argument passing sleeve. We can also **pass-by-request**. Here's an example...

```
//Inside EP1
requestForResX=context.createRequest("res:/X")

req=context.createRequest("active:lang")
req.addArgument("operator", "Code1")
req.addArgumentByRequest("operand", requestForResX)

result=context.issueRequest(req)
```

Just as with `addArgumentByValue()`, NetKernel constructs a PBV-space and places the request object as the representational resource.

Now when `Code1` sources its argument...

```
//Inside Code1
operandRepresentation=context.source("arg:operand", String.class)
```

..it causes the request to be resolved in the PBV-space. But this time because it was Pass-by-Request, then that request is itself issued and the result of that PBR-request is returned to *Code1* !! This is functional *lazy evaluation*.

You can see that the benefit of PB-Request is that the requested resource is not reified until the very moment it is actually required: **JIT state transfer**.

Declarative Functional Programming

You may find its not so common that you need to use PB-request from the NKF API, but you might like to know that the declarative request supports this too.

So you can readily construct declarative functional programs like this...

```
<request>
  <identifier>active:lang</identifier>
  <argument name="operator">Code1</argument>
  <argument name="operand">
    <request>
      <identifier>active:lang</identifier>
      <argument name="operator">Code2</argument>
      <argument name="operand">res:/X</argument>
    </request>
  </argument>
</request>
```

Which is your classic $f(g(x))$ construction.

By-Request arguments can be nested to any depth. This can be very powerful in the mapper, XRL, DPML, XUnit tests etc etc etc.

State Transfer: Does anything move?

Yet again we've covered a lot of ground. I'll leave you with one last thought.

I'm sure you know that Einstein conceived of relativity theory by posing the child-like thought experiment. **What would happen if I sat on a beam of light?**

I recall that when we were finding our feet with NetKernel and ROC, we asked ourselves a similar question...

What would happen if I sat on a request?

It's interesting that, like Einstein found, when you follow this question to its end you discover that it's a relative opinion of whether state is transferred or whether state is where it is and space bends!

In real-world systems you invariably find that a large part of the time you hit the cache. In such situations you can confidently say that state is not moving (its in the cache), only the contextual ROC space was bent for the given request. ♠

Resource-Oriented Computing and Programming Languages

4 – What is Code? What is a Function?

Peter Rodgers, PhD

I'm pleased to say that this series of articles has reached a point where we can think about some interesting stuff and consider the nature of code and functions in Resource Oriented Computing.

As a quick recap:

In the first article in this series I covered the basics of the ROC language runtime pattern and discussed how, by treating code as a resource, we get something we've called code state transfer.

In the second article I discussed the general nature of imports and surveyed a variety of languages detailing some typical models for accessing code resources referenced via internal import statements.

In last week's third article I discussed arguments and showed how, with the added dimension of spacial context, the difference between pass-by-reference and pass-by-value is only meaningful to the intrinsic state of the requestor, in the extrinsic ROC domain of a requested endpoint, PBR and PBV are exactly the same.

The story so far...

My hope is that these articles will have shown that ROC provides us with a computation environment where we are able to extrinsically apply computation to resources. And since functions are extrinsic, their computed resources are themselves extrinsic resources.

Caching and Performance

ROC provides an environment in which computation is normalized and extrinsic (relative to the address space). Furthermore being extrinsic, we can store all computed state. We can cache representations. The cache is extrinsic and is completely independent of language/code and even representational form.

ROC therefore provides an environment in which we can consistently and very efficiently determine when a given requestor may be served with the same stored state instead of executing code - we call this **extrinsic memoization** and I talked about it in depth [here](http://wiki.netkernel.org/wink/wiki/NetKernel/News/2/1/October_22nd_2010#On_State_and_Consiste)³. This property of the ROC abstraction has dramatic implications to the overall compu-

3 http://wiki.netkernel.org/wink/wiki/NetKernel/News/2/1/October_22nd_2010#On_State_and_Consiste

tational efficiency which I discussed [here](#)⁴.

So ROC is the definitive loose coupling in which architecture and code are independent degrees of engineering freedom. But this freedom and flexibility doesn't come at a computational cost. Quite the opposite. Strange as it at first appears, making computation extrinsic actually makes it more computationally efficient.

In a forthcoming article I want to talk about our motivations. Why we are doing what we're doing? What was the basis for even considering ROC?

To pre-empt that a little, when we started working on ROC ten years ago, my overwhelming concern was the brittleness and economic cost of software. I was an outsider provided with the best in conventional state-of-the-art software and it just didn't add up. The cost of building even modestly complex systems was dwarfed by the innate limits on scale and the mind-blowing cost of change.

With this perspective, for our first forays into ROC, to get a grip on the economics, I was prepared to trade performance for freedom and flexibility. It came as a bonus when we started to realize that the freedom of ROC's extrinsic computation was complemented by more efficient systems. *Talk about a double whammy.*

Today's Installement

So, lets get to the point, I don't want to spend any more time on the basics or the justification for this etc etc. If you've read the previous articles you'll have got the picture. Today I want to go to the next level. To consider what we mean by code and functions and to show how with a simple mental adjustment to our point-of-view we can see whole new ways of doing things...

What is Code?

We've seen that language runtimes accept code as transfered resource state. So far, we've discussed Turing complete languages and we've assumed that the code is written in a general Turing complete language. In this section I want to show that the language runtime pattern applies more generally and to show that it embodies a continuum which spans general code through to basic configuration.

I'll also show that there is a natural pressure which inclines us to travel from general purpose languages to domain specific languages and, in the limit, constrained resource sets.

But first, some pictures...

[ncy](#)

4 http://wiki.netkernel.org/wink/wiki/NetKernel/News/1/41/August_13th_2010#P_v_NP

Illustrated Lecture

It's not a coincidence that we released the image processing utilities this week. It was a trade-off for me: spend an hour implementing the library so that I can use it for the discussion that follows, or spend longer using words that would be less effective at telling the story.

Let's consider the `active:imageRotate` tool contained in that library. Its interface has this form...

active:imageRotate + operand@[...image to be rotated...] + **operator@**[...transformation...]

Here's a PNG image (lets assume its resolvable with the identifier `res:/nkimage.png`)



The `active:imageXXXX` tools take an operator argument which should be transreptable to declarative form like this...

```
<transform>
  <angle>90</angle>
</transform>
```

Internally the tools use an HDS tree structure to interrogate the declarative transformation.

HDS is our "house-red" representation model. It is flexible and means that we can have a range of transreptable representations all be valid formats. XML transrepts to HDS, but we can also use HDS forests, we could even provide, in the requesting context, a JSON to HDS transreptor.

We really don't care much about the data type of the declarative transform specification. All that the `imageRotate` tool internally cares about is that it has a node with a name of `angle`. It uses HDS's XPath selection syntax to find this node with `"//angle"`. It therefore doesn't matter whether the `angle` tag is deeply nested in an XML document, a JSON element named `angle`, or just a single node HDS with name `angle`. They're all good.

I want to return to the topic of the implications of ROC and extrinsic functions on our choices of representation type for our resource state. While you have ultimate freedom, since to NK a representation is just a POJO, I'll talk about how a good choice of represen-

tation model can help complement the innate flexible nature of ROC solutions.

You might ask, is the transform resource pass-by-reference or pass-by-value? Well, from last week, you know that the `imageRotate` endpoint doesn't care. Just so long as its argument can be transrepted to HDS and has the angle in it.

To illustrate, here's an example of a single identifier using a data: URI for the operator to specify the resource that is the 320-degree rotation of `nkimage.png`...

active:imageRotate + operand@res:/nkimage.png + operator@data:text/xml,<angle>320</angle>

Which gives us the representation...



Configuratin or Code

The *imageRotate* tools is a runtime that takes a declarative *operator* and applies the specified transformation to whatever image resource is specified in the *operand* argument.

For this brain-dead simple tool you wouldn't really describe the operator resource as **code**, you might be more comfortable describing it **as configuration state**.

Of course I'm setting you up. What's the difference between this tool and the general Turing complete language runtimes, such as `active:groovy`, we've assumed previously? Here's a table comparing the two..

Operator Argument Property	active:imageRotate	Active:groovy
Representation Form	Not defined. Can be anything transreptable to HDS	Not defined. Can be anything transreptable to ByteCode

Resource Constraint	an "angle" node must be located somewhere in the structure	must be syntactically valid Groovy.
Effect	the presence of the operator argument state determines the computation performed by the active:imageRotate runtime.	the presence of the operator argument state determines the computation performed by the active:groovy runtime.

The details of the internal representation form or the syntactic constraints of the transreptable representations may be different. But the key point is the last line. The effect is exactly equivalent - the presence of the operator state in the runtime determines the computation it performs.

So from the ROC abstraction's extrinsic point of view there is no difference. The two tools are instances of the same pattern.

And yet... I bet you feel that they're really not same sort of thing at all. How can they be?

Set up

You've been set up, and as we'll see in this section, quite literally.

Does the base identifier of a runtime, with no arguments, such as active:imageRotate have any meaningful sense on its own?

Put another way, does active:imageRotate identify anything on its own, without the presence of the specific arguments?

These are peculiar questions since you could never resolve that identifier (the endpoint has a grammar that requires both the operator and operand arguments).

So its not a technical question. Its a question about our understanding of Resources and the nature of computability!

Hold on tight. This is the bit where we do the mental leap.

My answer is yes the parameterless base identifier of a runtime has a meaning and expresses something valid about the nature of ROC. And we get there if we step away from the "tree" of code/configuration and step back to look at the "wood", the resources.

To get concrete, active:imageRotate is the identifier of the set of all possible image rota-

tions. Specifically since we have an internal `java.awt.Image` representation constraint - it is the set of all possible JPG, GIF, PNG rotated images.

Yes, the philosophers out there will see that it contains the identity set of all images having zero rotation. Take care with getting sucked into these analyses too deeply. You will break your head if you go any further down this avenue, and yes you will see that the set of rotated images is also in the identity set! So stop right now.



Bounded infinite sets are weird and the root of many many paradoxes. So you don't need to go too deep into making meaning from these set identifiers. My advice is to just glance sideways at them every now and again and let them sit in the corner of the room. But always keep in mind that they're really there.

With this advice given, lets just say `active:imageRotate` is the set of all rotated images and leave it alone.

Things get more comfortable if we progressively introduce and pin down the *arguments* and consider what their more specific identifiers mean. Things get real so fast that we can even draw pictures to help us think about them.

Here's a table that progressively gives meaning to the sets of resources that are expressed by `active:imageRotate` with the *operand* and *operator* arguments progressively introduced...

⌋

Identifier	Representation	Description
active:imageRotate	∞ no representation possible	The set of all possible rotations of all possible JPG, GIF, PNG bitmap images.
active:imageRotate + operand@res:/nkimage.png		The set of all possible rotations of res:/nkimage.png. <i>Note the representation could not actually be computed by the tool - but for illustration it can be represented as the black circle of diameter 121 pixels. The convolved view of all the rotations.</i>
active:imageRotate + operand@res:/nkimage.png + operator@data:text/xml,<angle>320</angle>		res:/nkimage.png rotated by 320 degrees.
active:imageRotate + operator@data:text/xml,<angle>320</angle>	∞ no representation possible	The set of all possible JPG, GIF, PNG bitmap images rotated by 320-degrees <i>Look I warned you - stop doing this. You'll break your head if you start thinking about infinite sets inside infinite sets. Put it in the corner out of sight.</i>
active:imageRotate + operator@data:text/xml,<angle>320</angle> + operand@http://www.flickr.-com/...	Finite representation not shown due to space limitations	The set of all images on Flickr rotated by 320-degrees <i>That's better. We're back on concrete ground - even though this is a very large set it's computable *and* safe to think about. They can all be cached too!</i>

So now you've warmed up. We can do the same for the active:groovy runtime and its Turing complete friends...

Identified	Representation	Description
active:groovy	∞ no representation possible	The set of all possible computations by the Groovy Turing complete language runtime.
active:groovy + operator@-data:text/plain, context.createResponseFrom("hello world")	Hello world	The String resource from the execution of the script: context.createResponseFrom("hello world")
active:javascript	∞ no representation possible	The set of all possible computations by the Javascript Turing complete language runtime.
active:xslt	∞ no representation possible	The set of all possible XML documents computed by the declarative XSLT language runtime.

Hopefully at this point you'll recognise that the stuff I said right at the beginning in the introduction to the series...

"Ultimately I hope to persuade you to look at the world as sets of information resources. Like Michelangelo releasing David from the marble block, information engineering is the art of applying context to release information."

...wasn't just grandiloquent claptrap. This is very analagous to what you do. The art of information engineering (aka software or programming) is to define the resource sets (get some granite - the set of all possible sculptures within that block) and introduce the constraints to reveal the representation state (sculpt it).

Cardinality

The reason you might feel different about `active:imageRotate` and the Turing complete languages is that it feels like there must be more resources in the language runtime set. After all I could write a script that does the same job as `active:imageRotate` - so the set *active:groovy* **must** be bigger than *active:imageRotate* right?

I said set theory is weird but they're the same size. The cardinality (a fancy set-theory way of saying "size") of the sets of resources computable by the two endpoints is the same: [aleph-0](#) - countably infinite.

If you followed the reasoning that said pass-by-reference and pass-by-value are the same. Then we can say that every possible representation we supply as state to either runtime can be written "in-line" as an identifier. The set of characters in that identifier is expressible as a very large integer. The integers have a cardinality of aleph-0. Therefore the

two sets have the same cardinality.

To the ROC abstraction they are just instances of a pattern and produce representation state that is extrinsically managed as efficiently as possible for you. We are language neutral - configuration and code are truly and fundamentally the same. NK treats the computed resource representations just the same.

Open Runtime Problem

Right we're all running out of steam but we just have to briefly cover one more topic. I've called it the *open runtime problem*. Actually this came up only a couple of weeks ago when I was discussing the Security considerations of WebSockets.

You must never put a language runtime out on the wild west frontier of an open web server. Here's just one example of why..

active:groovy + operator@data:text/plain,while(true){}

Turing called it the *Halting Problem*. Godel called it the *Incompleteness Theorem*. In ROC you might call them "Unreifiable Resources".

At an absolute fundamental level there are a set of unreifiable resources for any given language runtime. The problem is that you can't tell ahead of time if you might be attempting to reify one by inspecting arbitrary code in the runtime (that's Turing's halting problem).

The upshot of this is that you cannot receive external state and issue that state to a language runtime for execution. (Believe it or not that's what the recent CREST thesis proposed and did not even mention these implications!).

Your common sense already meant you wouldn't even have considered this as an application pattern. But I just wanted to make it absolutely explicit.

And Relax

Before you get all concerned I'm not saying that the language runtime pattern has a fundamental security hole. If the code resource is in your local spacial context you're as safe as houses. Just keep it in mind that for a Turing complete runtime, external code-state transfer is a no-no.

But I didn't say that about runtimes in general did I. As I intimated in the preamble, the runtime pattern is powerful and in the limit of considering really simple configuration state (such as the imageRotate tool) you can very readily introduce constraint boundaries that can inspect this state and determine if the resource that is being requested is reifiable (and sen-

sibly within the context of the application).

So, this is an interesting observation, as you constrain down the set of resources accessible by a runtime - you actually can determine if the resource is reifiable. The Halting problem is actually just a limit on a particular class of runtime.

So my warning is not necessarily that you should never use the runtime pattern on an open web server. Just make sure that the set of resource it exposes are bounded and consistent with your application model.

Oh, and of course, validation and constraint of state can be architected to be an *extrinsic property* of the address space too! ♠

Resource-Oriented Computing and Programming Languages

5 – Function, Sets and RoC

Peter Rodgers, PhD

In last week's fourth article I talked about the relationship between code and configuration. I demonstrated that to the ROC abstraction they are the same thing. I also showed how, when we think about sets of resources, we gain a useful perspective on functions.

Hopefully I gave a sense of ROC's innate language agnosticism. A valid solution provides the information we need irrespective of what the technical implementation is.

Turing Overkill

We also saw that there are compelling reasons to step back from Turing complete runtimes - not least in that a system composed out of configurable domain specific runtimes can be *verifiably constrained* and can have *provable integrity*.

For very many problems, starting off with the assumption that we must code in a Turing complete language is just overkill. You know the old adage: *With great power comes great responsibility*.

For systems coded with Turing complete languages you could rephrase it as:

With unlimited power comes unlimited responsibility.

Conversely for systems composed from extrinsically constrained configurable resource runtimes:

With constrained power comes limited liability.

Limiting Liability

The natural bias of an ROC system is towards limiting liability. The inherent risks and brittleness of *full-fat code* can be bounded and constrained and kept safely encapsulated within the black-box of Endpoints which, as the series has shown, are extrinsically composed with requests.

I'm not saying you never need to write low-level code. I'm just saying that in ROC it needn't be the default assumption and it's certainly not the dominant task. (*Made up number alert*) Typically we see that classical code construction is at most 20% of an ROC solution's development work.

Another day I'll talk about the practical consequences of ROC on the development cycle - for now assume 20% is relative to the total system development effort. ROC considerably lowers total system effort.

What I'm trying to convey is that ROC and its extrinsic model is an environment in which composition (structuring and mapping to extrinsic functions) and constraint (validation and introduction of boundary conditions) are what you spend most of your time doing.

**And* you gain the scope to introduce provability and quality assurance. *And* the economics of adapting to change are sane. *And*, as you're sick of hearing me say, it makes the system perform and scale better.*

**And* you don't believe me...*

Well fortunately just as I was writing this hysterical rant I got an email from Carl Conradie.

Carl works for a large insurance corporation and he's recently been self-teaching ROC/NK. His company has an existing portal and he decided to replicate it in NK. I gave him a small leg up by providing a simple bare-bones copy of the portal architecture we use for the NK services. This time last year it took me a day to knock that up (FYI it's about a 100 lines of module spacial architecture - no "code", all composition of the pre-existing pieces that come with NK), so bare in mind that I gave him a day's head start.

With his permission, here's what he emailed me yesterday...

*"I just spent the most productive 48 hours of development in the last ten years.
The penny has dropped!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!*

I just duplicated using NK (and your portal) what a team of 6 developers accomplished in 3 weeks using RAD and WS portal.

Ok and it's muuuuuuch faster."

I suspect you are conditioned to be extremely wary of blatant marketing hype like this. (If not, I know of a kind Nigerian prince who also corresponds with me and who I could put

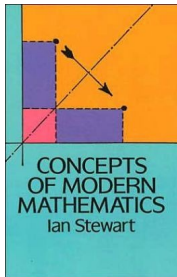
you in touch with).

However, if you followed last week's article, the fact that ROC allows you to think and work directly with sets of resources is the hard-core foundation to these practical values. As we've been building towards, and I hope to nail below, there is real substance behind my outrageous hype.

Today's Installment

For this week's sermon I'm going to push a bit further on our understanding of functions and from there try to offer a definition of ROC.

What is a Function?



"The concept of a function is a strong contender for the most important in contemporary mathematics."

[Ian Stewart, Concepts of Modern Mathematics](#)

Before you glaze over, this is not going to be a dry intricate examination of the mathematics behind functions (if that's what you expected [look here](#)).

Most of you, like me, will have had a technical education where at some point you were introduced to functions. You might have started with the linear function

$$y = m x + c$$

You almost certainly have a strong sense that in mathematics, functions are about the relationships between numbers - after all that's what maths courses spend most of their time concentrating on. Teaching you the tricks of the trade: adding, multiplication, division, drawing graphs, simultaneous equations, differential calculus etc etc.

As we learn computer languages we step away from mathematics to think about code and logic and variables, but still the basic sense of function we learnt in math(s) classes sticks with us. The functions we program take numerical values as their arguments, but we broaden it a bit and use String arguments and then broaden still further and have object references.

But ultimately, because we're programming on a Turing machine, all of these are still fundamentally relations between numbers (the state stored in the memory of the computer).

Some of you probably did some form of higher education maths course. This probably had some very dry formal definition of a function with terms like domain, range and target. (again look here for a recap). But, like me, you probably paid your dues, regurgitated the stuff in an exam and, if you thought about it at all, were satisfied that someone had bothered to put in place all this formal stuff but were pleased to move on.

I want to take you back to the foundations. I think we sometimes forget what we already know...

The Function Reloaded

Maths Symbol Reminder: When you see \in it means "element of". A membership relation between an object and a set. When you see \subset it means "is a subset of".

Right this is not going to be formal or complete, just a brief memory jogger and some pictures.

Firstly, the concept of a function is very general and also very simple. It can be stated in three lines. A function is:

1. Domain **D**
2. Target **T**
3. A rule that for every $x \in D$ specifies a unique element $f(x) \in T$

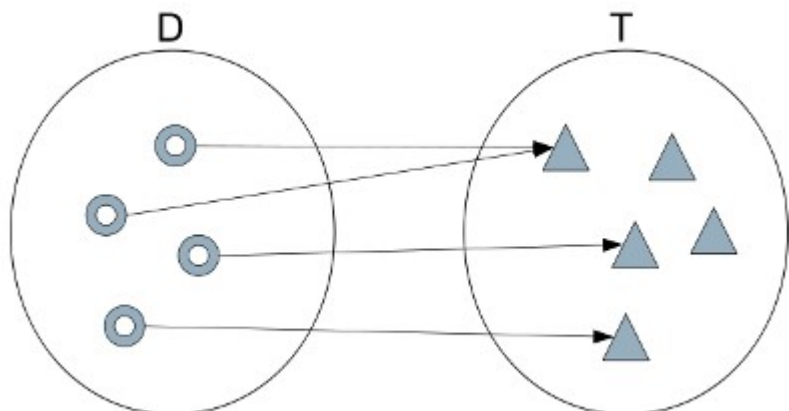
I said I'd keep it brief. What? That's not enough for you?

A function is a rule that takes a member of a first set (the Domain) to a member of a second set (the Target). "Rule" is a bit of a loaded term, so you can also say a *mapping (from D to T)*. The critical part is that to be a function, the mapping must be unique: x maps to $f(x)$ and nothing else.

It becomes a lot clearer with a picture...

The Function Concept:
The arrows depict the rule/mapping.

Notice how general this is.
We're not saying anything pre-



scriptive about the nature of the sets or their elements. It could be the set of circles, the set of points on the Euclidean plain, the set of bitmap images.

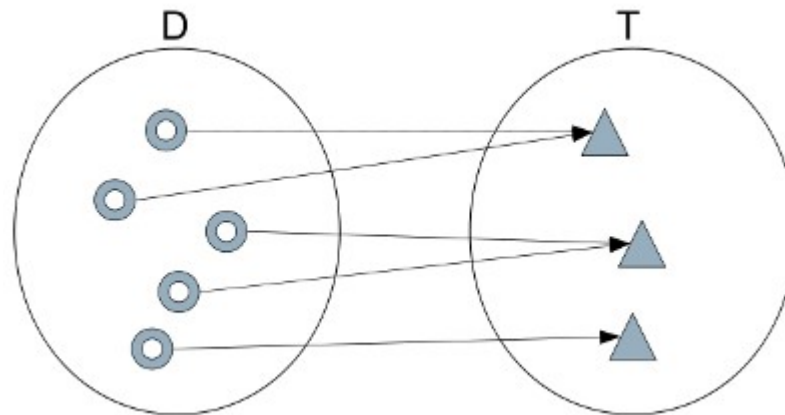
The reason we have a preconception about functions is that most of our maths training spends its time concerned with techniques for manipulating functions where D and T are the set of real numbers R .

Another interesting observation is that, the rule (mapping) doesn't have to be calculable. It just has to be valid in principal - it could be too expensive or just too hard to work out. Impracticality is not a limitation. We'll explore this a bit more later.

From the definition, represented in the previous diagram, it follows that there are some particular patterns which we can draw pictures of...

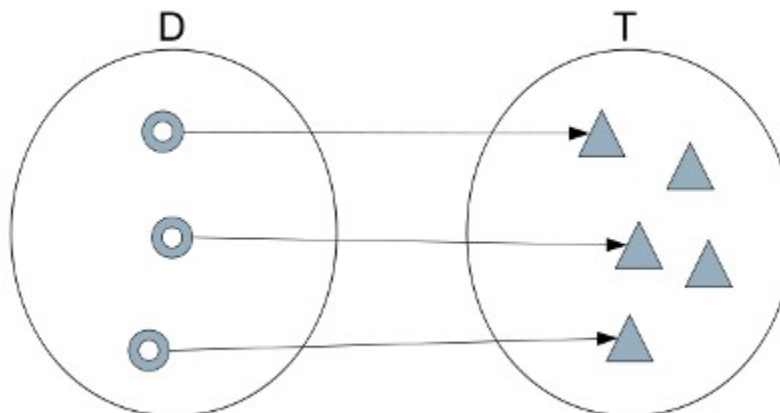
Surjection

When every element of T is mapped to by one (or more) elements of D the function is called a *surjection*.



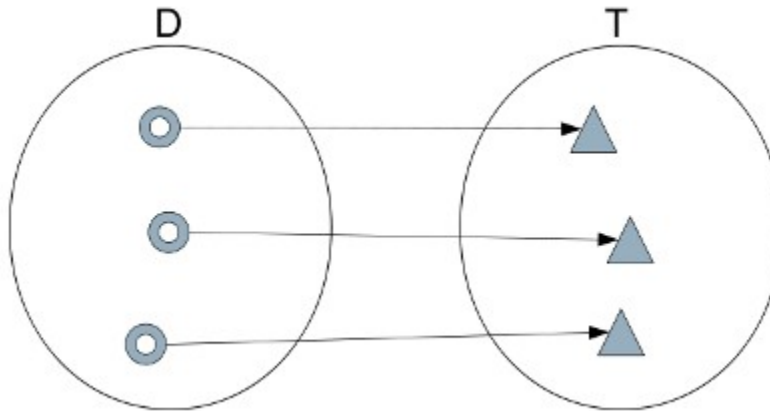
Injection

When every element of T is mapped to by at most one (but could be zero) elements of D the function is called an *injection*.



Bijection

When every element of **T** is mapped to by precisely one element of **D** the function is called a *bijection*.



You can see that this is the combination of surjection and injection. If you turn the arrows round you still have a valid function from **T** to **D**, which is called the *inverse* function.

Bijections are the functions you'll be most familiar with. Take for example $y = m x + c$ and the inverse $x = (y - c) / m$, given the x-coordinate of a point on the line you can uniquely determine y and vice versa.

Resources, Representations, Sets and Functions

Hopefully, you'll see that last week when I went off on my flight of fancy and started describing runtimes as bounded infinite sets I wasn't doing it for artistic effect. I was leading here and using the general set-based concept of a function.

Putting everything we've discussed together we can start to think about ROC using the core concepts of resources, representations, sets and functions.

What is the Web?

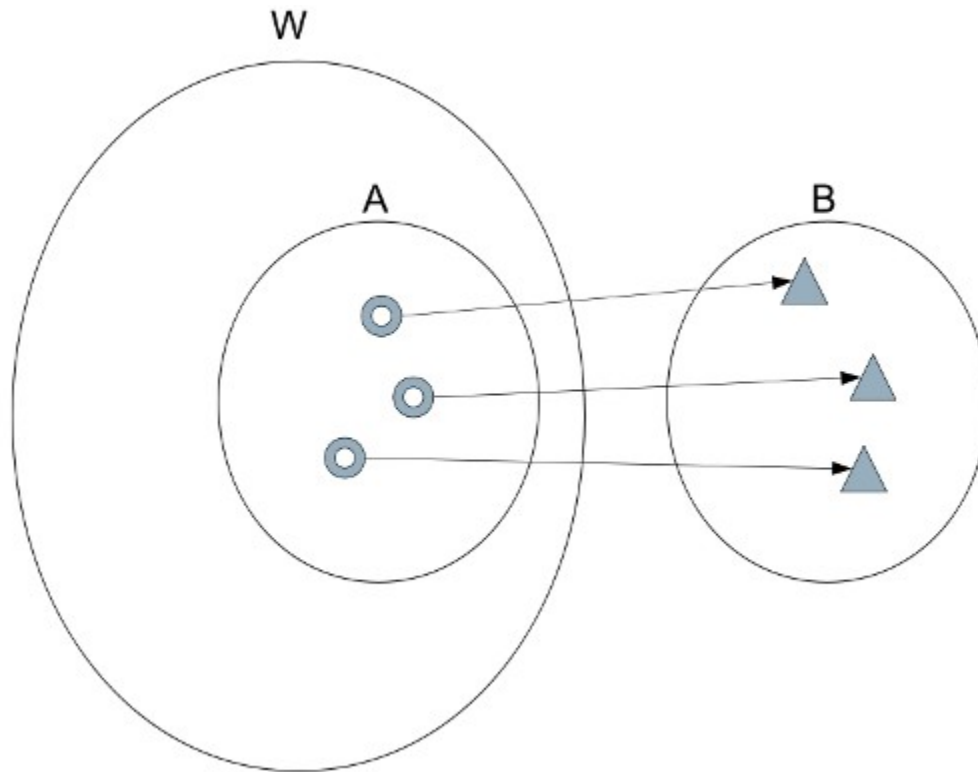
Let's warm up with a simple ROC system - a small subset of general ROC. Some people call it the "Web" and usually its described like this...

The World Wide Web, abbreviated as WWW and commonly known as the Web, is a system of interlinked hypertext documents accessed via the Internet. With a web browser, one can view web pages that may contain text, images, videos, and other multimedia and navigate between them via hyperlinks.

[Wikipedia Defintion](#)

But I reckon we can do better if we step away from the trees...

The Web (**W**) is the set of all resources with identifier beginning "http". A Web Browser (**B**) is the set of all possible representations of resources. A Web page or Web application is a function which maps a subset $A \subset W$ to **B**.



The World Wide Web: A Web-Page or Web-application modelled as a function of sets

Now obviously, I am not being rigorous here. I'm assuming an instantaneous steady-state model where time is not a degree of freedom. I've also rolled-up the functions whereby **B** combines the representations into one composite rendered representation (the pixels in the view port) - these steps can be shown consistently, but they'd make it harder to get a sense of the big picture (and we'll need this picture when discussing ROC below).

I'm also omitting the "user function" which is a "function that selects functions" (aka navigates, adds input etc I don't want to open that can of worms). You also know that Javascript is now essential to web apps and you can think of it as time-domain (or user-driven) changes to the rendered representation but since we've frozen the time and user dimensions we can say javascript plays no part in the instantaneous state view.

However, hopefully with these caveats and without formal rigour, I think it offers a useful perspective.

For example, we can see that this example diagram is modelling a GET function. We can also visualize PUT and DELETE as functions from **B** to **A**. And we can see that POST is a mapping from **B** to **A** followed by a mapping from **A'** (A modified by received state) back to **B**.

We can see that **W** must contain the empty set - the resource we obtain when there is either no server hostname resolvable by DNS or the server does not have a web server.

We can also see the point and very *raison-d'etre* of the W3C has been to ensure that there is one global **W** that is consistent and well defined. And indeed, perhaps with lesser success, that there is one **B** (well at least a platonic B with many instances) every browser offers (approximately) the same set of all possible representations ([see my article](#) on HTML5 for a discussion on the approximate consistency of B instances). So the W3C, have ensured that the Web has matured such that it is a consistent function - given x we *uniquely* get f(x) no matter where we are in the internet.

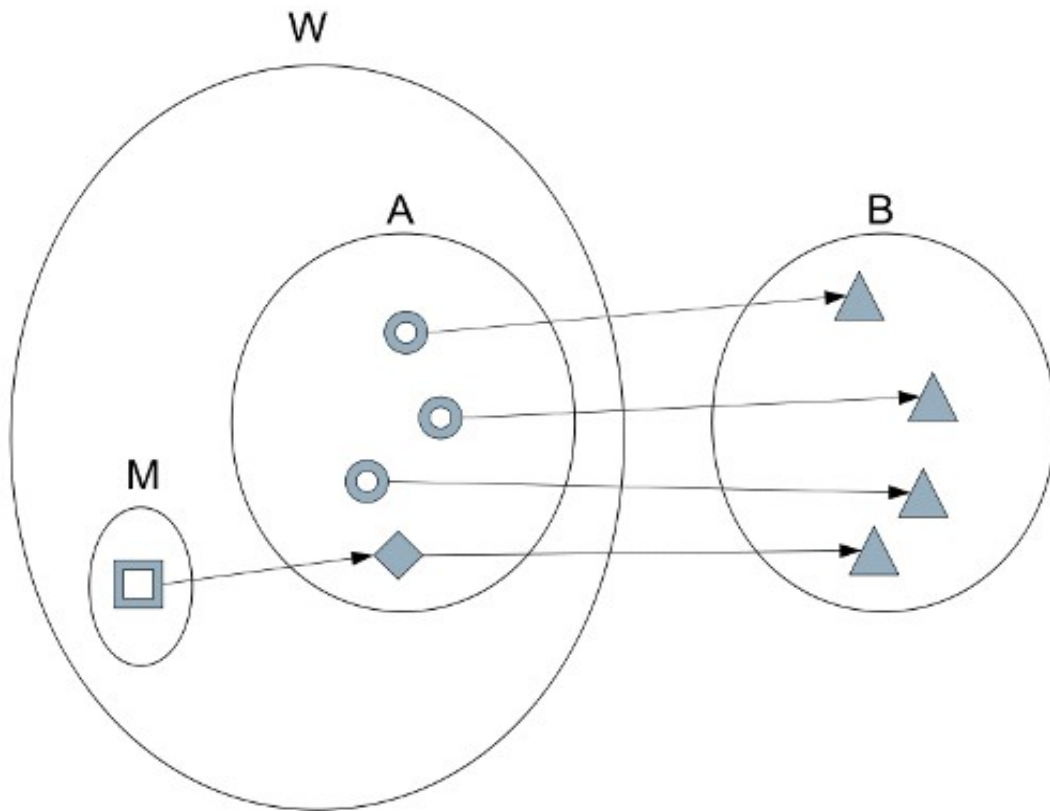
As I confess, I'm not saying this is formally rigorous. But you can see that this treatment could be formalized. We can, in principal, consider the World Wide Web as the set of all Web-functions. Fortunately, the practicalities of computability don't exclude this - as discussed earlier.

Composite applications

Is there a point to this? Am I just guilty of academic pontification? Hopefully not. I think this viewpoint gets to some deep truths. For example with it we can have a much better sense of architectural constructs like, for example, the mashup pattern...

Those who've been following these ramblings for years will know that I absolutely hate the term 'mashup'. Its totally dumb and unhelpful and I think reflects very poorly on the general understanding of what the Web is. I prefer 'composite application' - right glad I got that off my chest... (I have similar misgivings about 'cloud'... mutter grumble humbug...)

We can represent the mashup pattern as follows:



Mashup (composite web application)

We can see that a mashup consists of an injection of set **M** into **A** and thence to **B**. Where **A** and **M** are subsets of **W**.

What is ROC?

As I have said many times before, ROC is a generalisation of the Web. But I've been guilty of not sweating the details to justify this crazy sounding claim (*maybe its the physicist's psychosis "give me the general, spare me the details"*). But maybe at this point we at last have a shared context with which I can attempt to justify my assertion...

ROC poses and answers the question: Why have just one **W**? Why not have as many **W** as we need at any scale and granularity. Why not create **W** on the fly. And, while we're at it, why not generalize **B**, to the set of all possible Turing computable state.

So ROC is a construct in which applications are functions mapping from an N-dimensional generalisation of **W** (which we have freedom to define) to the set **B** of all possible computable representations. (*I confess I don't know how to draw a snappy diagram of this, however the new space explorer does give you some feel for it. However in the limit of a single space it collapses to the Web diagram shown above and with a self-consistency*

that doesn't require the caveats we gave for the Web discussion.)

ROC also introduces something new. A formal and consistent model for how we move from one **W** to another. In ROC the **W** are called **spaces** and the formal model is called **resolution**. We create applications **A** by defining function constructs that map from spaces to endpoints and so to representations ($\text{rep} \in B$).

You can now also see that the **grammar** is in fact a set description language. It enables the finite description of bounded infinite sets. (That's what we were doing last week when discussing the "interface" of `active:imageRotate`).

Furthermore, the **mapper** is a construct for logically defining functions (without the need for physical endpoint code). The mapper is general and permits us to define surjection, injection and bijection functions. (See why its called a mapper now? I've run out time but I was going to draw some diagrams of typical mappings and let you identify which type of function they implemented. I thought it would make a good game for your office party.)

With power over **spaces** and **mappings** we can implement simple composition patterns (internal "micro-mashups", spit!) but, unlike web mashups, we are not limited to injected subsets of a single space, we can construct elegant composite architectures of great sophistication and power spanning any number of spaces. (*Ask Carl Conradie about the Portal structure if you think I'm losing it*)

ROC allows us to step away from the minutiae of the "coding" and define broad extrinsic relations between sets. It follows that we can do more with less.

It also follows why ROC absorbs change. Change can be applied i) by introducing new sets with no side-effects on an existing solution and ii) by broadening existing sets whilst preserving legacy subsets. These factors alone shake the pillars of the economics of software.

We also gain a normalized view of all state in the system (we discover the members of **B**) and can consistently (due to the uniqueness criterion of functions) determine whether we actually really need to do any computing at all! (*It runs faster - really!*)

ROC is a generalisation of the Web. It may be a lot more besides, you tell me.

As I'll talk about next time, I don't think we've even started to scratch the surface of what it enables and what we'll discover... ♠

Resource-Oriented and Programming Languages

6 – ROC: State of the Union

Peter Rodgers, PhD

Given a certain information problem you have two choices:

1. buy a turn-key product, a solution that exactly satisfies your requirements, or
2. use a framework to build a solution yourself.

When I started seriously trying to do information engineering in the mid-90's I looked hard at the standard model (turn-key or framework). Whichever way you looked, I didn't see what I wanted. Either course felt like it had built-in limits, hard edges that when I reached them would leave me exposed to innate brittleness.

In my gut, I felt like the problems that had turn-key solutions and/or frameworks were small scale instances. There was something beyond them. Something more general and more powerful.

To use a horrible clichéd metaphor: information engineering is like bridge-building.

Image: Roman Arch: State of the art for ~2000 years.

You can cross streams and smallish rivers with a wood scaffold and stone blocks to put up a [Roman arch](#). But to bridge estuaries you need engineering models, cantilever moments, reinforced concrete stantions, suspension, box-girders; you need engineered architecture. And fundamentally you need in-



sight and grasp of first principles.

You could say that frameworks have advanced - they are more flexible. But to me there's a historical precedent...

It seems like frameworks today are analogous to when bridge construction first shifted from stone to wrought-iron. They had a new material, but they carried on thinking the same way. Look carefully at the picture on the right - yep, the world's first [iron bridge](#) was a Roman arch!



Image: World's First Iron Bridge - 1775: Iron Bridge, Shropshire, England

Pushing the metaphor, the history of engineering shows us its not sufficient just to change materials (languages, object frameworks, dependency injection...)

To make real progress you have to step away from the confines of the materials, to view the problem in terms of core principals. This is why I used the recent series of articles to show a progression from the novel, but familiar, language runtime pattern to sets and functions.

In ROC, first we consider the resources (fundamental problem) then we choose a language (materials: stone or concrete, steel or carbon fibre). Ideally we arrange it so that construction comes down to composing a solution from customized pre-fabricated units (box-girders).



Image: The most beautiful bridge in the world - 2010: Millau Viaduct, France (notice lack of arches)

As I say *ad nauseum*: ROC separates architecture from "code". What I mean is it is a

technological foundation that allows us to treat the first principles of the engineering problem, independently from the selection of materials and the mechanics of construction (languages, code and coding).

I hope that this year's newsletter articles have begun to shed light on what then happens...

Just as when the iron bridge builders began to think differently, so in ROC, by stepping out of languages, the *engineered information* solutions you can achieve can be breathtaking and beautiful.

Reality not Metaphor

This is a pretty limited metaphore. Information engineering isn't like bridge-building. You don't (and can't) always know in advance what the problem is - unlike bridge building, information engineering doesn't (and cannot) have tight specifications. Frameworks will help you build bridges but often in the real-world it transpires that the problem wasn't about bridges at all.

Ask yourself why large-scale government IT projects fail so often? I would argue the primary problem is one of ongoing change of requirements and partial specifications coupled to conventional software.

Information engineering has to be different to mechanical engineering. It demands an assumption of malleability. Rigidity and specification compliance must be introduced as extrinsic constraint.

Even here, there are lessons from physical engineering. Every mechanical/electrical engineer is trained to incorporate tolerance in a system (the famous: engineering factor of two) - conventional software is strongly typed and tightly bound. Rigidity is baked into the foundation, there's little room for tolerance.

And yet we instinctively know that to solve problems that you can't specify and/or will be required to adapt to future change, you have to look to emergently complex solutions. Biology is just such an information engineering problem...

You can think of biology as the information problem of sampling the environment with a sufficient Nyquist critereon, that the computational energy cost of information capture is less than the energy source (food) that is acquired. With the additional constraint that there is an energy surplus in order to avoid predators and to propagate the system's genetic information.

The diversity of nature readily shows that large scale sophisticated solutions are

achieved by progressive and incremental change.

To be discussed

This years technical articles have attempted to introduce, highlight and contextualize a bunch of the ideas that underpin ROC. I've been trying to gradually show that you can use NK with a "framework-mindset", but you can also, step away from the details and think about the first principals of the information engineering architecture.

You can just get things done with NK if that's all you need - but I think having an ROC perspective is the way to get to the next level.

After last week's article (part 5 in this series), I received some really detailed feedback from Jeff Rogers which I want to incorporate in an article soon. There is still a ton of things I want to discuss and I'm aware there are several topics that are still only partially covered...

- Thinking and modeling in terms of sets of resources.
- Choosing Representation models - thinking of representations as sets.
- Constraint and confidence boundaries.

Our Possible Futures

This state of the ROC-union address is at a point in time where we're between the past (defining ROC, making it real with a hard-core implementation) and the future (the consequences and opportunities engendered).

What are our possible futures? I think you can see a number of orthogonal axes...

1. End-user solutions

- I showed that the Code state transfer pattern has a trend line: going from Turing complete languages to configuration state to constrained resource sets. This line doesn't stop. It can be projected forward to a point where it is safe to give an end-user (a call center operator, an account manager etc etc) a palette of click-fit tools and they can do their own "programming".
- This isn't a vision, we've been progressively putting in place the core infrastructure for this over the last year and a bit. (Remember all those - "we added metadata", updates).
- We are bringing this together now and plan to have some really exciting stuff for you by the time of the NetKernel conference in April.

2. Space Runtimes

- You learn the runtime pattern by coming at it from a classical scripting point of

view. But you saw that the pattern holds even if the code-state transfer isn't "Turing code".

- The next step for this is to have a "Space Runtime" - that is a runtime which receives a declarative spacial architecture definition and instantiates and "executes" the spaces.
- Surprisingly perhaps, we're actually doing this pattern inside a number of the existing tools that come with NK; but we're doing it down on the metal. The next step is to abstract it and to provide a declarative specification language.

3. Multi-dimensional "General Web".

- This year we snook out the NetKernel Protocol (NKP). It allows NetKernel's general ROC abstraction to span servers.
- As you saw last week, ROC is a generalisation of the Web.
- With NKP it is valid to think of a multi-dimensional distributed Web address space. The "General Web". (No-one can accuse me of not thinking big).

4. Extrinsic Data

- I've been showing how ROC is innately about inverting the intrinsic nature of classical monolithic software. To have an extrinsic view of functions and services and the computational state.
- This trend doesn't have to stop. A natural progression is for the internal resource state to be exposed externally to the outside world.
- This is where the "Semantic Web" is coming from.
- I don't necessarily see RDF as the only representation model. Extrinsic data can be expressed in any suitable form: CSV, XML, HTML, JSON, SQL...
- However I do recognise that the principal of extrinsic data is inevitable. (I think the next topic also makes it more likely too...)

5. Contextual Confidence - Trust Boundaries

- With ROC you have direct control of the context within which information is accessed.
- Control of context is what traditionally gets lumped under the label "security".
- I don't think we've tapped the surface of the implications of ROC on defining trust and confidence boundaries in information systems.
- This is stuff that really does need you to step away from the code and think about the information resources and context.

6. Malleable Discoverable Computational Systems

- ROC is a dynamic computational system.
- You can see, in for example, the dynamic import pattern, dynamic state may be used to define spacial architecture
- The natural progression of this is to have state that enables higher order system capabilities to be emergently created.
- I can see that the core pieces are already in place for patterns of emergent, "self-generated" software... ♠

ROCIing the Cloud

Peter Rodgers, PhD

Maybe its because I've been tied up in contract negotiations all week. Maybe its because I'm a nerd. Maybe I'm just dissatisfied with the software industry's mystical faith-based approach to ascribing value.

Whatever the reason, I decided I really wanted to have a true engineering model of a hard core cloud-based architecture. And, of course, since this is what I implicitly espouse every week, to understand the affect of using NK/ROC versus a classical software solution.

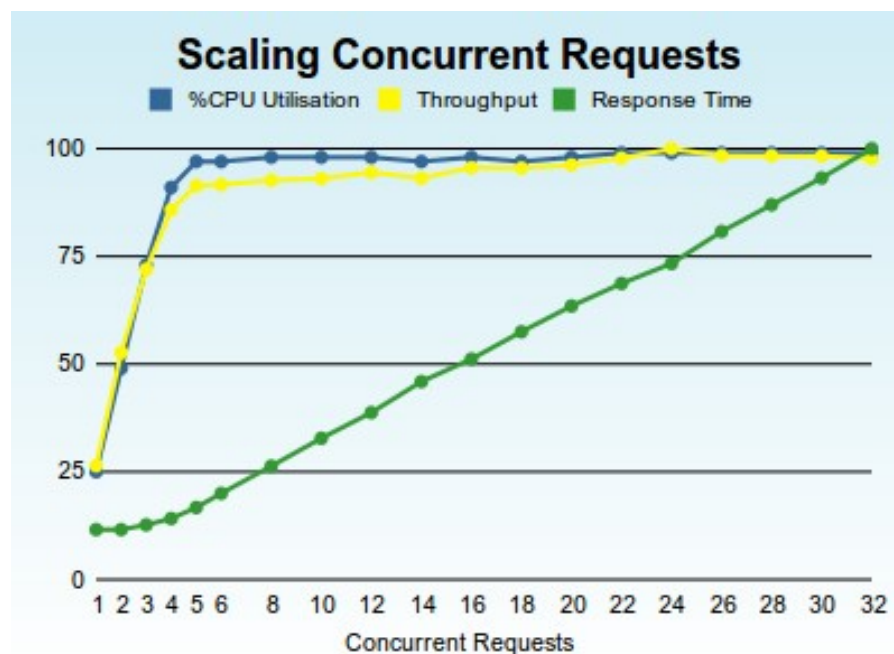
Measure Twice to Model Once

First we need to measure some real systems (*once a physicist always a physicist*)...

You might have seen this graph (below) before, its the results of running the *nkperf* scaling benchmark on my Toshiba Intel dual-core hyperthreaded laptop. The test software stack is NKEE 4.1.1, on Sun Java 6, on Ubuntu 10.04 LTS 64-bit.

Important points to note are that NetKernel scales throughput linearly with CPU cores when the concurrency of requests is \leq the available cores. Once we occupy all the cores, there's no more overhead to process with (more threads can't do any more work if there's no more CPU cores to run them on), so the throughput flat-lines and the response time linearly increases with concurrency

Its important to note that NetKernel itself (and the Java/OS software stack) continues to remain linear and displays no "drooping tail" in the throughput - the system is maxed-out but NetKernel maintains the throughput at the constant peak.

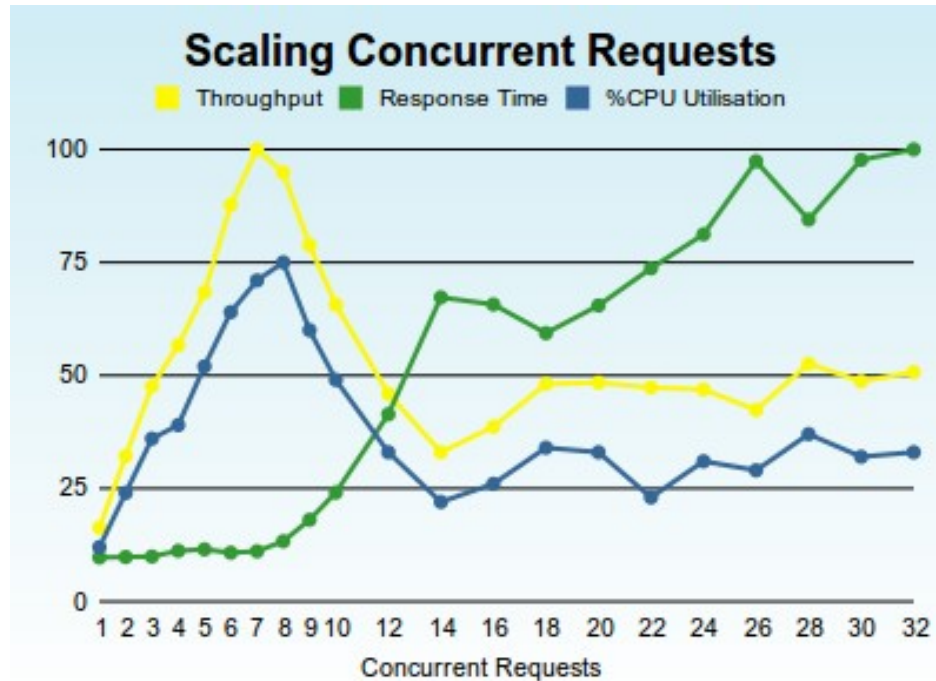


(Incidentally its interesting to note that my system has 4 logical cores but this consists of two "real cores" and two hyperthreaded logical cores. Notice that the response time shows this and starts to increase from a flat response curve when concurrency reaches three to four. This shows hyperthreading is good, but its not quite a true core)

Cloud Server

So lets try the same experiment on exactly the same software stack, but this time lets use a fairly high-end virtualized cloud platform with 8-logical CPU cores on a well-known public cloud platform. The graph belows shows nkperf running on the same NKEE 4.1.1, Sun Java 6, Ubuntu 10.04 LTS 64-bit but on a virtualized hosted server. (For reason's of respect for confidentiality we'll not reveal the platform).

We see that, as before, NetKernel scales linearly when requests \leq available cores. But notice that once we max out the available cores, we see a dramatic non-linear response as the concurrency exceeds the capacity. We know from our first results on a physical server (and repeated on many many customer server stacks) that this is not due to the NK/Java/OS software stack.

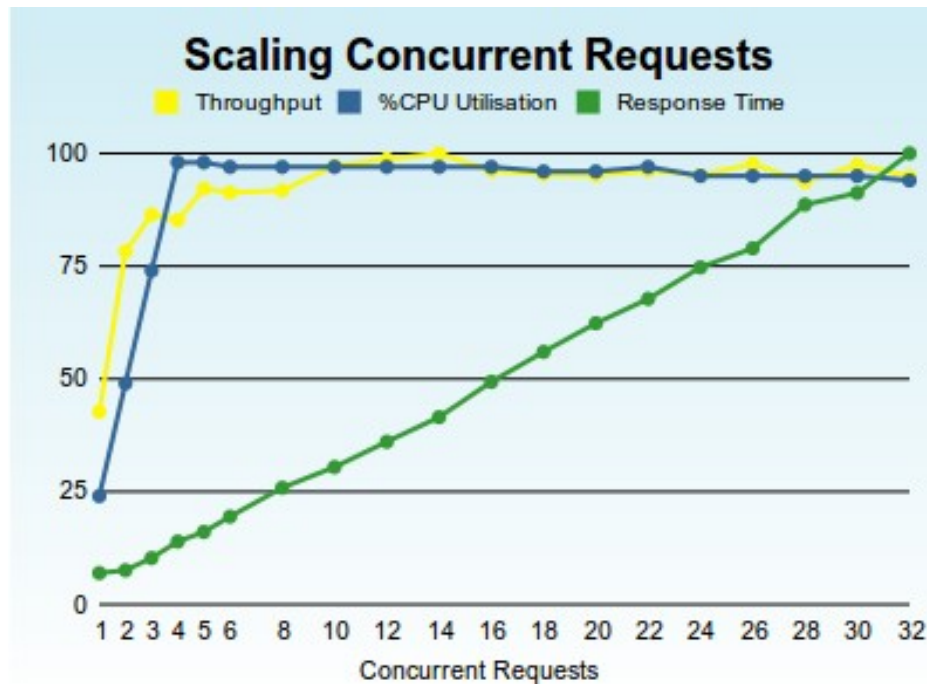


We can only speculate that this is either the underlying virtualization technology, or the platform's own extrinsic scheduler throttling. Given that throughput eventually settles at a suspiciously tidy 50% of *maximum*, my inclination is to think this is deliberate load management by the cloud provider for this class of instance.

Linux KVM Virtualization

We can estimate if this is a correct assumption by changing the virtualization technology. Here's the results when we run exactly the same software stack but this time on my laptop but on a KVM virtualized Ubuntu 64-bit with 4-logical virtual cores. (This is exactly the same laptop as for the first graph of physical cores and no-virtualization)...

It's very interesting that this virtualized instance shows qualitatively the same results as the physical results showed before. But we note that there are some tangible differences due to virtualization: we can see that the Intel virtualisation instructions are not supported on logical hyper-threaded cores, since we see our linear scaling stop at a concurrency of two. However we still see a solid non-drooping maxed-out throughput response curve - indicating that the intrinsic scheduling of a virtualized CPU is linear on the Linux Kernel's KVM virtualisation technology ("big up" to the Linux possy).



Whatever the underlying cause of the vendors cloud non-linearity, we can definitively state one firm conclusion: **Not all clouds are created equal; some are more equal than others.**

Clouded in Thought

OK we have some data, where does this put our modeling? Well the first thing we can say is that if you are running a free-running software stack with a thread-per-request (a la PHP, Tomcat and all the other standard J2EE application servers) then you are by definition operating at the far end (right-hand side) of the concurrency/throughput curve. This holds no matter what the underlying software architectures - if you have more threads than CPUs then you're not getting more work, you're just dividing CPU capacity more. This is the engineering equivalent of throwing spaghetti at the wall and hoping that it'll stick.

It follows that if you are running a regular software stack on the cloud platform shown in the second graph, you are operating at precisely 50% of the possible maximum performance of that instance. Or put it another way - in a large scale cloud architecture you're paying twice as much for cloud server instances than you should be.

There's another consequence of thread-per-request spaghetti slinging. Every thread will require that a substantial amount of transient memory will be allocated for its operational state (objects are created and the memory is occupied even if the threads are not being scheduled). So admitting every request to a system on its own thread means your host memory allocation has to be large, a lot larger than is actually necessary for the work being done.

As with cores and performance this is another lever that cloud platforms use to weight the pricing model - cloud memory costs rise non-linearly.

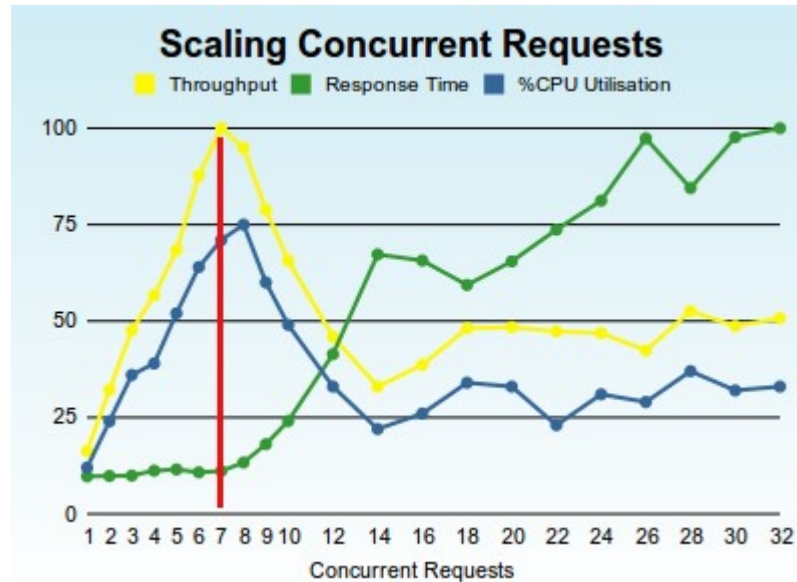
Software Load Lines

If you have a platform who's response curve is non-linear and which displays a clear peak in throughput it would be a very good idea to constrain your system to sit on that point. This is not a new concept in engineering, in fact in electrical engineering its called the [Maximum Power Transfer theorem](#). A variant in electronics is the design of the biasing of transistor circuits by consideration of the [Load Line](#).

What's needed in software systems, especially cloud-based systems, is a **maximum throughput load line**.

In the case of the cloud system we measured, you want your system to sit on the red-line...

One interesting observation, notice that peak throughput occurs at seven concurrent requests. Its interesting that we've consistently seen that these peaks occur at one less than the number of cores. It seems like you need to allow one core for the use of the virtualized operating system - not really a surprise but worth remembering.



OK. So how do we do this. Well with NetKernel its trivial. You just wrap your application in a throttle...

```
<overlay>
  <prototype>Throttle</prototype>
  <config>
    <concurrency>7</concurrency>
    <queue>1000</queue>
  </config>
  <space name="Wrapped Application Space">
    <!--Your app imported here-->
  </space>
</overlay>
```

Those half-dozen lines of "code" will lock your application load line on the peak throughput. Or to put it another way, that small change will double your system throughput

under load. Or put another way you can manage the same load on a NetKernel server as on two-load-balanced cloud servers with conventional software.

One more thing to consider. If you're only admitting seven concurrent requests into your application, and in this example you're holding up to 1000 in queue, then your transient memory requirement through operational object state is 7/1000ths of an unmanaged system. Or put it another way, you definitely won't be visiting the premium-priced high-memory-end of the cloud provider's platform smorgasbord.

OK, so we've identified one dimension in which NetKernel and ROC can kick-ass in the cloud. Lets look at another...

Systemic Caching

There's a simple statement that you've probably heard me say many times before. **NetKernel caches everything.** But what I've neglected to do is explain what that actually means in terms of performance and system costs.

Firstly, lets just explain why NetKernel's cache is unique. After all, the world is currently awash with vendors offering magic-bullet caches for your standard applications.

In a classical software system, you introduce a cache by binding it (either explicitly or implicitly with a point-cut) at a tier in your architecture. You write your application to manage that reusable object state at that single cross-cutting boundary in your architecture.

This is a one-dimensional cache architecture. No matter what your system is doing, there is a single dimension at which you have decided you will pro-actively manage computational state.

NetKernel is different. NetKernel caches everything in every tier. It has a multi-dimensional systemic cache. It allows the actual value (the amount of use (hits) vs the true computational cost of the state) to determine what state is worth holding on to. Not where it comes from in the system architecture! If something is computationally expensive and is being used frequently you'd like to eliminate the cost of recomputing it - no matter which tier of your architecture it is in.

This is what NetKernel does.

I won't go into the additional features that NK offers with its systemic dependency hierarchy to atomically and automatically manage cache consistency. Instead, we want to understand what the effect of caching has to our system's throughput and how it affects our cloud platform engineering model.

Realworld Hit Rates

The fundamental reason that caching works is because the real world obeys natural statistical distributions. Or in cloud software system terms: the same stuff gets requested more than once.

The degree to which stuff is re-requested depends on the application. But for typical modern web applications and services its common to have power-law distributions. In more classical problems its likely these will flatten to normal-distributions. But generally caching stuff pays off because the physical world is consistent and behaves predictably.

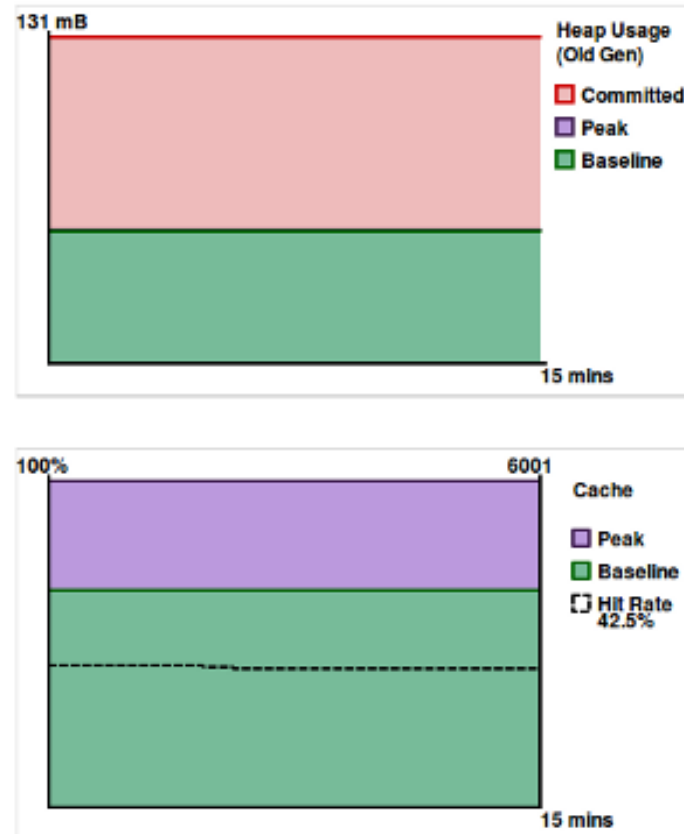
OK, so we need a concrete measurement again so that we can consider this in our model.

Shown below is a snapshot, taken earlier today, of the memory and cache charts from one of our standard NKEE servers. Its actually this wiki's host NK instance (where you're reading this news letter). This NK instance is also running the [1060 research](#) corporate web applications, the [NetKernel portal](#) with the NKEE support and other portlets, the [bugxter](#) software management system, the [NKSE download](#) services, and a sundry collection of various REST web-services.

As of time of the snapshot, this NK instance had been up for 1 week 6 days 23 hours. The last time it was rebooted was due to the system-core update we shipped two weeks ago. FYI we always deploy and test any updates on our real production machines before we release them to you guys.

OK, what is it telling us. Well it showing that our memory footprint is within sensible engineering limits - this JVM has a heap of 196MB and you can see that the actual Old Gen is 131MB of which our footprint is occupying no more than about a third. Or put another way, our memory allocation is three times our minimal needs.

This system is throttled and so, as I explained above, our peak transient memory use is always bounded and within engineering margins of control. It doesn't really need me to say it, but this is proving what I implied earlier, NetKernel is incredibly light on memory use. Which in our model means you won't need to pay for expensive cloud memory.



OK, look at the second chart showing a snapshot of the cache state. This system has a cache size of 6000 (peak). We see that we are typically churning 30% (baseline) - ie this is stuff that has been determined to be insufficiently valuable. (As a short aside, this is another reason why NetKernel out-performs ordinary application servers. We are holding all transient state and determining if its useful retrospectively. It follows that we dereference and release garbage in a predictable way and which is very sympathetic to the underlying JVM garbage collection. From an operational state management point of view NetKernel is the JVMs perfect OO application.)

Hit me baby one more time

OK so lets cut to the important part. Notice the dotted line and the number 42.5% in the chart legend. This is the cache hit rate and is the instantaneous percentage of requests which are being satisfied from cache.

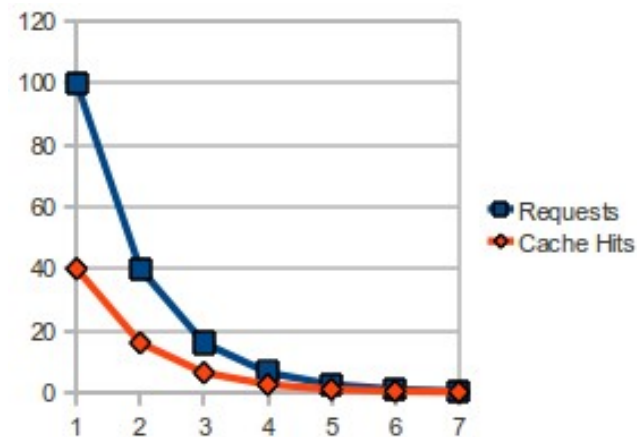
As I said above, applications vary and indeed application loads vary with time of day/week, but the real world can't be fooled, it'll keep showing up with statistical distributions. I can say that across all our production systems we typically see production cache hit rates of between 40% and 50%.

On some servers, like for example the documentation server, we could provide a large cache and get a 100% hit rate. But we tend not to do this, since the engineering trade-off is: why waste the memory (pay for memory) when the main "users" of this site are the almost constant web-crawlers. Our doc server runs on a 96MB JVM and last time I looked its cache hit rate was running at 30% - which ain't bad when you think that web-crawlers have very poor statistical distributions.

Hit Rate Significance

So we have a measurement, and a sense of what a production system hit rate might be. But what does it actually mean?

The diagram below lets us think about the effect of hit rate. Lets imagine we have 100 requests in our system and the instantaneous cache hit rate is 40%. It follows that of those 100 requests, 40 will be satisfied from cache. It further follows that in the time saved not computing those 40 requests, we can compute 40 more requests. But of those 40 requests 40% of those will be cache hits saving us 16 requests. We can do 16 more requests, but 6.4 of those will be cache hits etc etc...



The diagram above shows the requests and corresponding cache hits for a 40% cache hit rate.

Expressing a cache hit rate as a percentage is not immediately intuitive to our human brains. But what it is really, is the common ratio of a geometric progression. To determine how many extra requests we can process because of the savings of the cache hits we need to sum the geometric progression.

If you remember your undergraduate maths classes you'll recall that the sum of a diminishing geometric progression is

$$S = 1/(1-x) \quad (x < 1)$$

where x is the common ratio. (In the case of 40% hit rate, x is 0.4)

OK so now we know what cache hit rate actually means. Here's a table of hit rate versus relative efficiency. Where no cache (0% hit rate) is 100% relative efficiency - ie we have to compute all the requests which is the same as a system with no caching at all.

Cache Hit Rate %	Relative Efficiency %
0	100
10	111
20	125
30	143
40	167
50	200
60	250
70	333
80	500
90	1000

Again, our brains aren't good at intuiting these things, but we can see that hit rate actually corresponds to an exponential relative performance gain.

Getting concrete, our own typical systems have operational hit-rates of 40-50%, what this actually means is that they are 166%-200% more efficient than an ordinary system.

But there's another fact that we ought to also take in. NetKernel is caching in multiple dimensions, this is not just about single architectural boundary layer. It is also weighting the cache to hold the most valuable state in the entire system. It follows that a cache hit is dis-

proportionately effective in NK (compared with a typical single-dimensional caching boundary). Because ROC systems are dynamic equilibrium, we can't really say how much additional efficiency this translates to. But what we can say is that this projected relative efficiency is the worst case benefit. A real NK system is doing better, a lot better, than this.

But we need a model, so we'll take this base worst case improvement as our "cache-benefit factor".

Cost Model

OK we have two sets of measurements and some understanding of the factors at play. We can now build a model of what this translates to in terms of TCO of operation of a cloud server instance.

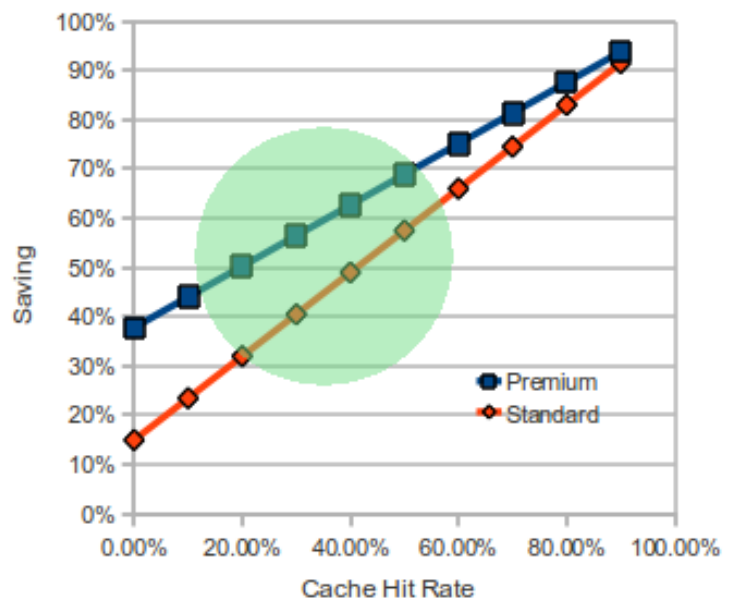
Lets combine the two effects. Lets assume we go to the "extreme effort" of adding the six lines of code to introduce a throttle. That alone gives us 2x throughput gain on the vendors cloud platform. Now lets consider our typical operational cache hit rate, lets take the lower number of 40% hit rate, which corresponds to a 1.66x throughput gain.

Combining the two gives us the **NetKernel Factor = Throttle Gain x Cache Gain**. In this case that's 3.33. Or put another way, a NetKernel instance with a 40% hit rate on a cloud server will have a minimum performance equivalent to 3.33 free-running classical software stacks on the same platform.

Now let's see if we can translate that to something we can understand financially. For obvious reasons we can't reproduce other people's published pricing for their cloud instances. But lets say a standard cloud instance costs something like 25c per hour, a higher-end model with more memory costs something like 75c per hour. Lets also include the NetKernel Enterprise license cost - which when broken down from an annual rate, is about 30c per hour. How much would using NetKernel save just on cloud operational costs.

I have a spreadsheet with the hardcore calculations which I'm very happy to give to you - just drop me a note. But here's a useful ready-reckoner...

The green region signifies the "zone of probable operation". ie a likely typical systemic cache hit rate for everyday applica-



tions.

So what does this actually mean for our concrete example. Lets say I have a 40% cache hit rate and my system is a hard core highly utilized server on the cloud platform. You will save, just on operational cloud fees, \$6,402 per annum for standard instances, and \$23,515 on the 75c per hour premium instances. (This is after you've factored in the very reasonable pricing for NetKernel).

But even if you're sceptical on the effects of the systemic ROC multi-dimensional caching (which I deal with below). Look at the 0% hit rate on the cache. Just by shaping the load line with NetKernel's separation of architecture from code, you are still making significant cost savings.

Finally, there are hidden cost savings which are not included in this model. The model doesn't include the reduction in operational memory requirements (a non-linear cloud weighting factor), nor does it include the actual instance storage costs. Many cloud platforms actually back-load their cost model so that storage is not part of the hourly operational rate - if you use NetKernel you have fewer instances and so smaller ongoing storage costs. Finally, cloud providers also charge for network transfer costs. NetKernel not only caches as much of your application as it can, it also automatically deals with 304 Not-Modified responses - so most web applications, even if they are dynamic, will operate with automatic minimisation of network transfer costs.

Computer Science

Its easy to be cynical and look at the numbers I've quoted above with a degree of scepticism or downright mistrust. However I hope that by now you've realised that I'm at heart a scientist - it goes against everything I believe in to not try to present an objective and impartial perspective on things.

So having constructed the model above I got to thinking if maybe there was something hard core that I could present to give you something irrefutable and reproducible for you to judge my assertions by. Then I remembered this article on the [Ackermann function](#).

For those without the computer science background - the Ackermann function is a pathologically extraordinarily expensive computation which is not just exponential in time, but tetrential. To see how bad it is - the value $A(3,12)$ can't be computed using a local intrinsic recursive Java function, as the JVM runs out of stack.

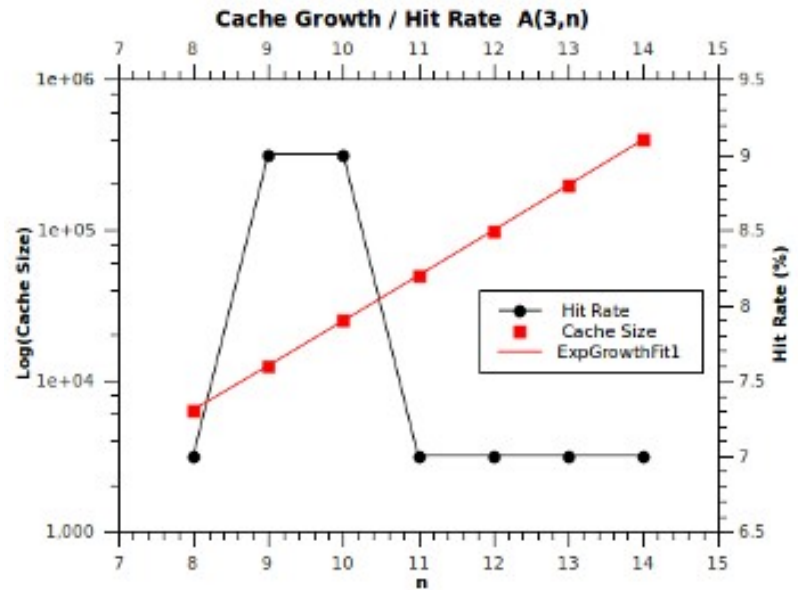
If you read my previous article, you'll see that it explicitly shows how NetKernel is able to discover by extrinsic resource oriented recursion repeated state actually inside the Ackermann algorithm's resource set. For example, take a look at the [visualizer trace](#) of the $A(3,6)$ to see how NK keeps biting lumps out of the recursion cost by rediscovering existing system-

ically memoized (cached) state.

This got me thinking. What is the cache hit rate of the Ackermann function when computed on NetKernel? So I loaded up the demo and tried computing various $A(m,n)$ values, each time clearing the cache and then after each run looking at the cache hit rate (as reported in the [representation cache viewer](#)).

Here are the results for the $A(3,n)$ series...

What we find is that, as we'd expect, the cache size grows exponentially, however rather amazingly, the hit rate remains roughly constant at 7%. That is, NetKernel is discovering that 7% of all computations in the classical Ackermann function are not necessary as they're points that were already known. Now this is even more amazing when you realize that each point hit saves exponential time cost!



Effectively, NetKernel is biting exponentially large chunks out of the computational state space. For free, automatically!

So now you might say, well if its doing this surely it must have a measurable effect on the computation performance - yeah that's exactly what I thought too. So here's the execution time (plotted on a log scale to show the linear exponential gradient).

So the amazing thing here is that NetKernel computes the Ackermann function faster than a local recursive Java implementation for $A(3, n>9)$. But recall that its logarithmic space shown on the graph. The difference in the gradients of the two lines shows that in the time-domain, NetKernel is exponentially faster than the local recursive Java function! Our only problem is that Ackermann is still going exponentially faster than NetKernel - so we can never catch it up. (In other words NP is not P).

And my point here is this. Nobody would suspect that a pure recursive function would contain hidden statistical distributions of reusable state. You'd never even try to memoize the Ackermann function - in fact it would be very hard to do it since we've had to use a recursive identifier notation just to write down the name of the Ackermann values (eg $A(2,A(3,2))$ see for example the visualizer traces).

So my impartial objective statement of fact is this. When I say NetKernel discovers the multi-dimensional statistical distribution of reusable state within an entire application (not a one-dimensional layer), then I am in fact telling the absolute and provable truth.

It follows that it is also true when I say. NetKernel is damned efficient. Really really efficient. And that means my assertions about TCO hold water. Or, to put it another way, even after you've paid for NetKernel licensing, if you compare the costs of using NetKernel with a so-called "free software" solution - then the free solution is more expensive.

Final Word

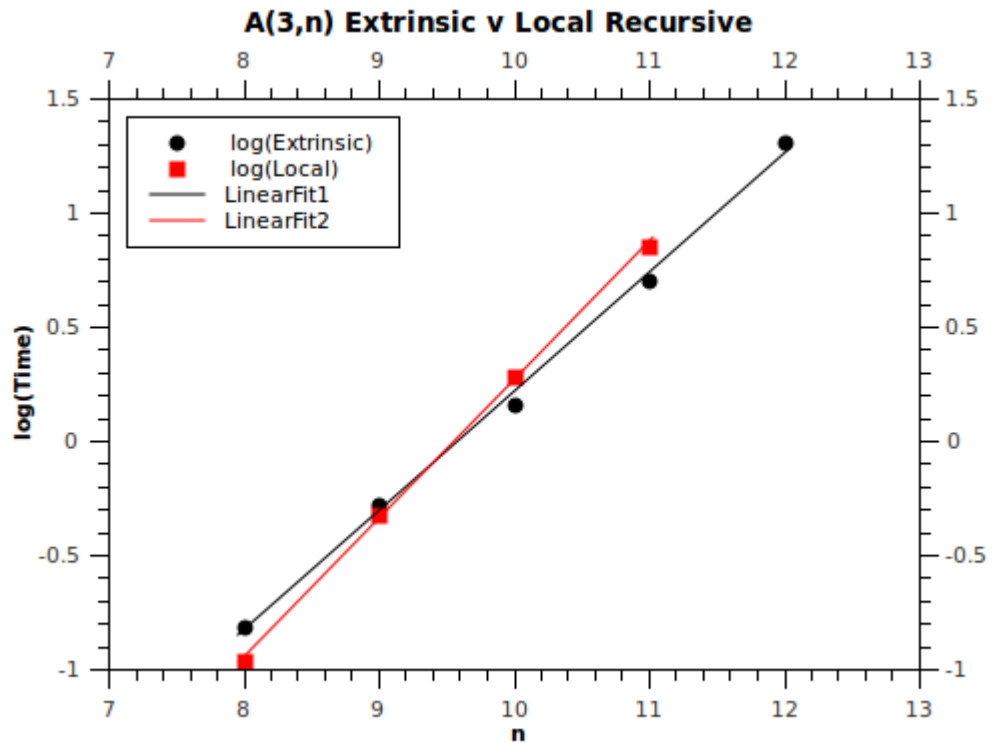
Of course the frustrating part of this entire article is that its carefully focused on just the tangible operational costs of software choice.

But the reason we embarked on the research behind ROC back in the late 90's was that it was the economics of the development and long term evolvability of classical software that made no sense.

NetKernel brings the Web-inside not for some fancy technological or computer science reasons. But because the Web has amazing economic properties that ought to be applied to software engineering at whatever scale.

In the Web the cost of change is less than the value added. How do I know? Because the Web has grown exponentially since its birth.

So even though there are operational, performance and just plain good and sensible engineering reasons to use NetKernel. The one that I can't show you easily is the most important one of all. With NetKernel it takes fewer developers, to build larger scale systems, that are more robust and more malleable with respect to change. ♠



ROC and data structures

Peter Rodgers, PhD

When I first started to think seriously about software, I was naive and would ask innocent questions of my more experienced colleagues. Question's like: "What's the best general purpose data structure to do such and such?"

I'd come from the world of Physics and my expectation was that there would be a body of evidence with world-relevant selection criteria based upon rigorous engineering evaluations. Like, for example, when you are required to choose a type of steel cable for a suspension bridge, or an op-amp for a circuit, or even just surface mount resistors - you have a body of principles, guidelines and best practice and this eliminates the need for trial and error. I was expecting something similar for data structures.

I was therefore somewhat surprised to discover that software seemed to be much more free and easy. It seemed like "you make it up as you go along" was an acceptable answer. I never could reconcile this and, for the most part over the years, I have bitten my tongue and avoided conflict.

Well not today. Today I'm coming off the fence.

Objects Objects Everywhere

[Coleridge](#) would have put it better but here's a verse from "Rime of the Ancient Coder" (which I did just make up)...

***Data, Data every where,
Miss it if you blink,
Objects, Objects every where,
Nor seldom stop to think.***

This poor verse is an attempt to lightheartedly consider the reason for the standard practice I observed.

When the *arrays* and *structs* of C became encapsulated as the internal members within objects

we were "liberated".

It no longer mattered what structure the data had - it was inside the object and, if you were obeying OO convention, the object's methods were how we interacted with it.

Of course at some point we are always required to touch the structure - in the return types of the methods. But the temptation (nay the express goal) of object oriented design is that a method itself returns another object. As that nice Prof. Feynman was told in his [public lecture by the old lady](#): "Young man, surely its turtles [object's] all the way down".

It's easy to understand that when your language is object oriented and your paradigm is to create classes of objects within that language, then its easy to see why the structure in which you place your state is no longer regarded as a first order design criterion.



But in ROC we are forced to confront headlong the structure of data. Since in ROC we have taken a radical step. We have broken free of the "language-plane" and determined that transfer of state* is the overriding design consideration of a system. Therefore our choice of state bearing "representation-structure" becomes rather important.

**Actually minimisation both of computation and of state-transfer is our real systemmic goal. Since these cost energy. So lets state Rodgers' Razor: When two information systems, given the same inputs produce the same outputs, the one that requires less energy is better. This is followed by the conjecture: It is also the one that will take less energy (time, effort) to evolve and adapt to a change in the information system.*

The Tyranny of the Bean

In the object oriented paradigm it seems like such a natural thing to do... you put your state inside a value object and you provide getters and setters to interact with that state. That was how it was. That was the natural thing to do... Until it became a problem. A big ugly problem...

Once upon a time when threads existed but processors were single cored - we had the illusion of parallel processing - but our systems actually ran step by step. So you could share objects and know that you could always synchronize on the object - knowing that the overall system would not be materially affected by single-track access to it.

But today the myth of parallelism is no longer a myth. Its real. To make full use of the cores in your CPU you must aim never to have to synchronize code. Synchronization is the death of scaling.

But all is not lost. *To eliminate synchronization we just need to treat state, and state bearing objects, as immutable.* If we share state and no-one can change it, then we don't have to synchronize when using it.

So now you see the problem: **Getters good. Setters bad.**

OK, go and look at a class library. It doesn't matter which. I bet its littered with mutable objects. Just look at any of the core java.* APIs. The overwhelming convention of the last 20-years has been that in designing objects they will have "state reading methods", but also a liberal mixture of state mutating methods and sometimes (often!) you even find read methods that cause state changes as a side effect just for good measure! (I'm pointing no fingers - but I've sure kissed a lot of toads).

Starting from Here

Of course we rarely start from a clean slate. We will generally have existing code and hence existing libraries of objects. And of course nobody is saying you can't use objects nor that any system should not be able to incorporate existing investments. But clearly we need a few guidelines that allow legacy objects to "behave nicely" in the highly concurrent, stateless world of the resource oriented domain.

The Rule of Immutability

The most basic rule of thumb is that when writing any endpoint you must never make calls on mutating methods of any representation that you may be processing. Every representation is highly likely to be shared. You have absolutely no right to modify its state in the physical domain of code. State changes can only be allowed to happen "in-band" via requests in the logical ROC domain.

Here's how to think of it. When an endpoint is invoked it does not receive any transferred state (it is something of a fundamental misconception in REST that "state is transferred" - it is not, in a Resource Oriented system it is context which is transferred). Rather, an endpoint is provided with a context in which the state it is informed about becomes accessible.

Practically speaking, it is wise to envisage that all state is static and it sits in the central state space (aka the cache⁵) and that your execution is just a context in which you have been

⁵ I dislike the word cache. Its such a dumb and overloaded word. What NetKernel really has is a common state

given an object reference to that state. So now you see why it would be unforgiveably rude to mutate the state.

Pseuo-immutable Interfaces

The rule of immutability is hard to police and, given the twenty years of free-living mutable OO-modelling we've enjoyed, hard to break the habit of. So there is a simple way to permit the use of legacy object models in the ROC domain. Add a level of indirection.

Provide a wrapper class which first and foremost implements an immutable interface. (Due to the pragmatics of existing object models, you will also probably inevitably have to provide a separate mutable interface, exposing the dangerous mutating methods. But by keeping the two separate you ensure that a developer is required to make a deliberate decision to move from immutable to mutable access).

And now you have a much simpler way of writing your endpoints. Instead of asking for the raw object type. You always ask for the immutable interface. (You can always provide transreptors that will automatically get you from raw to immutable forms⁶).

Example - IXDAReadOnly

Here's a concrete example. A long time ago, in what now feels like another life, we realised that `org.w3c.dom.Document` was an absolutely terrible representation. It has the classic mix of mutating and readable methods. It requires state management to navigate its structure - even when the structure can be expressed in vector form (XPath). It has even been known to have non-thread-safe read methods etc etc.

So we wrote a wrapper object model to make DOM more suitable for ROC. Its called IXDA and can be found in the `xml-core` module - its a semi-declarative approach to API design (more on which later).

The most important observation is that it separates access into two interfaces. There is a primary pseudo-immutable interface called `IXDAReadOnly` - its methods look like this...

```
package org.netkernel.xml.xda;
import java.io.*;
```

space. This state is the realization of representations of abstract resources. So my preferred term for this state space would be "The Realization". It constitutes the measured approximation of reality in which the system is currently executing. "The Realization" is managed in such a way that the state it contains is qualitatively the best ongoing approximation to the real world that it can maintain - which is what I mean when I constantly bang on about thermodynamics and minimising energy and entropy.

6 Of course this is not truly immutable, its a form of pseudo-immutability and its just a way to keep everyone honest. Kind of like agreeing to drive on the left (or right, if you're one of those funny lot who drive on the wrong side).

```

public interface IXDAReadOnly
{
    String eval(String aTargetXPath)
        throws XPathLocationException;
    boolean isTrue(String aTargetXPath)
        throws XPathLocationException;
    String getText(String aTargetXPath, boolean aTrim)
        throws XPathLocationException;
    IXDAReadOnlyIterator readOnlyIterator(String aTargetXPath)
        throws XPathLocationException;
    void serialize(Writer aWriter, String aTargetXPath, boolean indent)
        throws XPathLocationException, IOException;
    void serialize(Writer aWriter, boolean indent)
        throws IOException;
    IXDA getClone();
}

```

And, for pragmatic reasons there is a mutable interface that looks something like this...

```

package org.netkernel.xml.xda;

public interface IXDA extends IXDAReadOnly
{
    void append(IXDAReadOnly aSource, String aSourceXPath, String aTargetXPath)
        throws XPathLocationException, XDOIncompatibilityException;
    void insertBefore(IXDAReadOnly aSource, String aSourceXPath, String aTargetXPath)
        throws XPathLocationException, XDOIncompatibilityException;
    void insertAfter(IXDAReadOnly aSource, String aSourceXPath, String aTargetXPath)
        throws XPathLocationException, XDOIncompatibilityException;
    void applyNS(String aTargetXPath, String prefix, String uri)
        throws XPathLocationException, XDOIncompatibilityException;
    void removeNS(String aTargetXPath, String prefix)
        throws XPathLocationException, XDOIncompatibilityException;

    //etc etc etc
}

```

Any endpoint that wants a quick and easy semi-declarative way to interact with XML resources can request an IXDAReadOnly representation. They are then able to safely access the state. If they subsequently need to construct a new XML data representation they have the ability to immutably clone the original and perform modifications to the clone. Or indeed create a new empty instance and immutably "pour in" parts of the original.

This is rather an extreme example - a complete abstracted model for interacting with and manipulating XML resources. But the principles will hold for your own domain specific object models. Provide a pseudo-immutable interface. Write endpoints to always request the immutable interface. Provide transreptors if necessary.

Sometimes (honestly, quite often) you will have to provide a wormhole in your wrapper to get out the underlying POJO. When you do this it is always a good idea to make the developer sign in blood that they have taken on the responsibility for not mutating the state. We do this by the simple expediency of ending our method names with "ReadOnly". So you would provide an interface like this...

```
CustomerRecord getCustomerRecordReadOnly();
```

Meaning: ok, you can have the underlying CustomerRecord - but you *absolutely promise* not to change its state. **This is READ ONLY!**

In summary, follow these basic guidelines and you'll suddenly find you don't need synchronization. Your solution will scale. And you know what, you won't need to resort to learning that next magic bullet language that everyone is raving about.

Starting from There

So now let's imagine a pristine world with no legacy. We are immediately faced with some requirements. Our data must be in a form which is easily consumable by many possible endpoints. In ROC - computation is kept separate from state. State is transferred to code to create new state (*No its not. I told you nothing is transferred but context - but if it helps to think of it that way then go ahead*).

We also know that inherently, there are, relatively speaking, small sets of core information. But that most systems involve the composition of that core state (and combination and transformation of those compositions, and of those compositions, and of those compositions.. ad infinitum). It is the nature of the real world that it necessitates lots of compositions (some would say: mash-spit-ups).

We intrinsically know that it is impossible to perfectly model the world - and, it is infinitely "more impossible" to model the future state of the world. (Think of the map maker who set out to create the perfect map. It would be 1:1 scale! He ran out of paper and got into an infinite recursion trying to put himself on the map).

Because we cannot model the world with perfect fidelity we have to anticipate compromise. We have to expect that our representations won't always have what we need - but maybe if we can combine and transform ones we do have we can get what we need. In fact the real world is often satisfied by just such emergent combinations.

The ability to discover and evolve emergent state is an innate goal and satisfiable expectation of ROC. (It is, for the record, also why OO systems are brittle and eventually break and have to be thrown away. Inheritance does not a composite make.) It is also the basis by which we allow an information system to provide serendipitous emergent information - stuff no-one anticipated ever thinking would be important. It is new information that leads to new opportunities which leads to new value which ultimately leads to your company being more successful than its competitors. (Every business no matter what it does is an information system).

So, let's take a stab at answering the impossible question: What is the best data structure?

What is the Best Data Structure?

First of all some ground work. In ROC we are first and foremost conceiving of resources as abstract sets of information. Nothing says that those sets cannot be represented in countless concrete representations. I am not saying one form is better than another - there is no such thing - they are the same resource. What we are aiming for is some engineering principles. Some guidelines that when we need to build something we can use and avoid the pit-fall of trial and error.

Our first requirement is that whatever we choose we need it to be composable. That is we need to be able take one or more representation and relatively easily combine them into a new representation. So what we're aiming to get an engineering handle on is, what does "relatively cheaply" mean and how do we know it when we've found it?

What are our Choices?

At the highest level, the possible data structures are as follows: atom, list, array, tree, graph (and map - to be discussed later). Each can be used to represent a set.

It is pretty easy to show that any of these data structures can provide the basis for a general purpose computing system.

For example - assembly language is dedicated to computation using atoms of state. (by atom we mean bits, bytes, integers, floats and even strings). But we all know that its pretty tricky to do composition with atoms without inventing higher order data structures. Actually its completely possible - as is shown with the Turing tape. Its just a bit "fiddly".

Equally, we know that the next highest structure, the list, can also provide a perfectly adequate structure. LISP.

If you've done any engineering mathematics you'll probably have used MatLab. This takes multi-dimensional array processing as the basis for an effective computational model.

OK - how do we find Goldilocks? Who wants to try some porridge...

Atom

Too small. Too fiddly. Move on...

Wait for it! How will we know when we've found porridge that's just right? First we'd better think about what our taste in porridge is...

Our Taste in Porridge

Taste is subjective. What follows is led from experience and from hard won knowledge of balancing trade-offs. There are no rules. Anything is possible. This is my subjective perspective...

A function (endpoint) is a mapping from one set to another.

$$S_1 \Rightarrow S_2$$

Or usually we write

$$S_2 = F(S_1)$$

Now think about an evolving business. It demands that S_1 must evolve - new information is added to the set all the time. How do we preserve the fact that in order to tolerate change F must still return S_2 (the layer that uses S_2 expects that S_2 is stable for their purposes) and we really really don't ever want to have to recode F .

So what we desire in our data structure as a set representation is this:

$$S_2 = F(S_1')$$

We desire that S_2 is stable under change of S_1 without needing to recode F .

Lists do not have this property since F would need to know the original length of the list. What happens if we append at the front rather than the tail. Or in the middle! So to keep S_2 constant we are forced to modify F to F' to work the same with the new S_1' .

Just as for Lists, so too for Arrays. They have the same problem of insertion and extension as Lists and so F must be changed to preserve stability of S_2 under change to S_1' .

So jumping ahead, what about graphs? Graphs are very easy to combine - they have the excellent property of being trivial to create set-unions (as Brian Sletten is fond of saying: "they can be slapped together"). Intersections can be computed relatively cheaply too.

But graphs do come with their own overhead.

For a start they do not have an origin. They demand that F must necessarily search to start its computation to find the S2 we desire. This means that while they can tolerate S1 changing to S1' - the day-to-day overhead of working with a graph is somewhat expensive computationally.

Furthermore, since even in a directed graph we can have edges that form loops that can return to the starting point - this means we are required to hold state as we traverse the data structure. Think of it like this - you will waste less time if you roll out a ball of string with you as you explore a maze. The necessity for the string is your "state overhead" for exploring the graph. (In ROC we are trying to minimize state - aka energy - remember *Rodgers' Razor*?)

But looking at the system in the large as an engineering problem, I would also argue that graphs are intellectually quite challenging for average developers. They're hard to think about and so hard to code against - which makes it error prone. They are intellectually expensive for the users and programmers of a system.

We intuitively know that writing code is easier when iterating over lists. Not walking graphs.

But graphs certainly have a role to play - graphs are powerful and indeed necessary when the outcome of the computation is not rigidly defined. That is, we are desiring to discover based upon a set of values and or relationships (known edges in the graph) the subset of the graph that satisfies some set of properties. This is the raison d'etre of the semantic web movement.

Shake My Tree

So what about the 80-90% case of stable repeatable systems with known resources?

Step forward Goldilocks - the Tree. A tree has the wonderful property that provided S1 evolves to S1' with an enlarging operation (adding branches, leaves) then in fact S1 remains a perfect subset of S1'. That is S1 is undisturbed by growth and evolution. Subsets are immutable under change.

Furthermore, computationally F can be written to use simple linear vector notation in order for a developer to apply computational focus to the state (subset) that they care about.

These vectors are called XPath's. In general an XPath that satisfies S_1 will also satisfy S_1' .

Finally, conceptually XPath's allow coders to move from "tree thinking" to "list iteration thinking" - which is natural to programming languages and is least error prone for developers.

In addition traversing a tree does not require any additional holding of state. The current position in the tree is sufficient state to unambiguously navigate down (and up, via parent relationships). You're never lost in a tree.

It follows that Trees have a very valuable property. They allow S_1 to evolve and grow without requiring any change to F . That is

$$S_2 = F(S_1') : \text{if } S_1, S_1' \text{ are trees}$$

notice we are not saying S_2 has to be represented in a tree - a resource can have any representational structure. All I'm saying is that to guarantee S_2 is stable without the desire to modify F and without knowing how the world might change, then my experience is that choosing tree structures turns out to be a good engineering decision.

Dimensionality

So the next question is - can you convey the resource information you need in a tree? Well of course that's a stupid question - if an atom works so does a tree. But is it the right sort of thing? Does it feel "roomy" enough?

A much better and precise way of saying this is: Does it have the right dimensions? Too few and you're restricted and processing and evolution are hard. Too many and you're too flexible, growth is easy but processing and human conception becomes hard.

According to the Hausdorff measure of dimension: lists have dimension 1 (too few), graphs have dimension N (which is down to your choice of graph and is often too many). While trees are in the goldilocks zone. They're about 2.x dimensional (cauliflower is 2.3, broccoli 2.6).

Which in short, is me saying: They seem to be roomy enough - but not too roomy. Not too hot, not too cold - just right.

NetKernel's Use of Trees

This discussion is just a public exhalation of something we we have known for a very long time. This is why the module.xml is a tree structure. Its not to make it hard to write and ugly (counter to some popular opinion). Its because we cannot anticipate the future. We need a

reliable evolvable data structure to represent the spacial architecture of the ROC domain.

We also use tree structure as the representation of choice for configuration state. That is state that is generally repeatedly used by an endpoint rather than which changes each request. Using trees for configuration means that the ROC tool set has remained incredibly stable over 5 generations of physical embodiment. Requests that "ran" on the first prototype NK back in 1999 will still give the same resources today.

The module.xml is a tree structure. The fact it's XML has the elegant benefit that we can allow configuration state to be expressed in-line too. It is reliable, evolvable, stable and tolerates ongoing and progressive change. So configuration state can appear in situ (no matter what new tool we invent) and, increasingly and very importantly for the future, it allows anyone to put their own extensible branches in (called metadata). Metadata driven architecture is already apparent in the RESTOverlay (Which has (apparently!) been regarded, as an impressive feat of engineering - not really, the fact we are using composite trees meant it took just 2 days work - start to finish), and is seen in the metadata driven nCoDE tools... but, we've barely scratched the surface of what's to come.

HDS

Our engineering need for the tree is seen in the only true native resource object model in NK. In layero there is the HDS object model. IHDS is immutable out of the box. It has the very nice property that it can be used to represent an atom, a list, a tree, a forest (not discussed but obviously if trees are good - then forests (multiple trees in a set) are good for similar reasons).

When you see XML configuration - we actually transrept it to HDS and then, if necessary, into a localized and specialized low-entropy POJO (cached so one time hit with optimal lookup).

In addition HDS also has the powerful ability to be inverted. That is, if we know the value of a node but we need to locate it in the tree. The HDSInversion converts the tree to a node-value \Rightarrow tree-node map. This provides $O(\log)$ time entry into the tree for the most efficient execution of a computation F on the contextualized tree when a value is already known.

The Bigger Picture

You know that I've got a bee in my bonnet. I'm pretty sure that ROC is about as efficient a computational model as you can have. If that's the case, then it ought to be present in other systems. Systems where the imperative (survival) demands that any extraneous energy use would be severely punished (death and annihilation).

I told you that I had a recent excursion into the world of biology. Well similarly I went looking in other places too.

The Language Instinct

For the longest time I've known that our predominant form of written communication is in tree structures (we call them books or documents, or even web-pages). This isn't coincidence. Its got to be because our brains have a resonance for tree structured information.

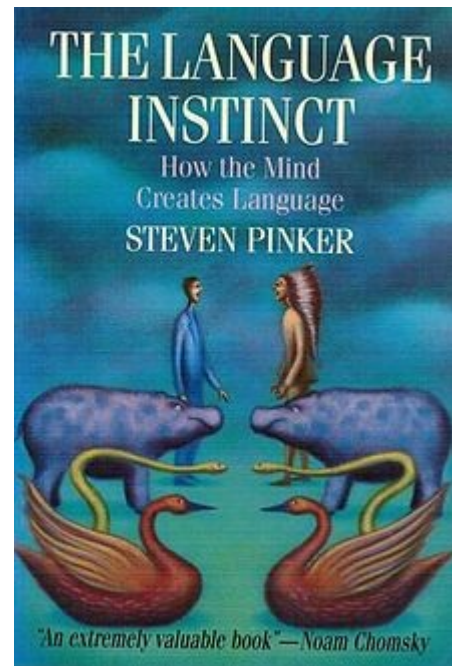
So armed with a pre-conviction that I would find things I recognised, I started wondering about language. Human language. The primary medium of exchange of information between humans.

I bought a copy of ["The Language Instinct" by Steven Pinker](#).

It's a beautiful book. Its the sort of stuff that should be standard knowledge, taught as soon as possible in school to every kid. Today's syllabus - Maths: this is how we count. Language Principles: this is how we all communicate. English (or your native language) - this is an instance of Language Principles. (I felt let down that no-one had told me this stuff earlier).

Guess what?...

- **First revelation**, the current standard model of the mind is called the "*Representational Theory of Mind*". It is discussed on pages 73-82. It is very very familiar. Its premise is that we (the human mind) resolve identifiers to abstract resources and reify these to representational state. (I am not kidding). I expressed it more succinctly since it is a little less concrete than our own world view - not surprising as we have full control of the internals of such an ROC system - the social scientist has to measure by indirect means and is prevented by ethics from taking people's heads apart bit by bit. But nevertheless it is uncannily familiar. (So now I need to go and talk to Social Scientists as well as Biologists - I need to get out more, twice as often).
- **Second revelation**. Every sentence you ever uttered (wrote, spoke) is a tree structure.



- **Third revelation.** Our brains have an abstract tree structured grammar model - this accepts infinitely complex variability in sentence structure to be "transrepted" to a normalized universal human information model. This theory (reinforced by repeated scientific evidence) by Chomsky is as important in its way to the human race as anything in the 20th Century. (Why didn't I know this before?)
- **Fourth revelation.** Every word you ever spoke is also a tree.
- **Fifth revelation.** These properties are universal - they appear in every language of every people of the world, and, for example, they also appear in sign-languages.

No matter what the language, both the sentences and the words are the input (and outputs) of our language processing system(s). They are both infinitely variable and yet, they (we) have the extraordinary property that when we use them, the same abstract information is proven to be conveyed from one information system (you) to another (me).

This is one heck of a strong piece of engineering best practice to underwrite the case for tree structured information architectures.

Back to ROC

Remember I said I went exploring language armed with pre-convictions. Remember I keep saying that REST paths are ok - that the NK simple grammar can parse them. But if you are building something to last and evolve you should use active identifiers and the active grammar?

Guess what? The active URI is a tree structured identifier. The active grammar is a tree structure normalizer so that abstract identifiers are decoupled from the evolving labels (identifiers) we give to information resources.

Do I have to draw a 1:1 map? Is it not evident we are standing on the shoulders of giants. Or maybe, like the map maker, we've been recursively standing on our own shoulders all along. ♠

On Metadata

Peter Rodgers, PhD

Its time for me to face my demons. Its time to discuss metadata.

Strange choice of introductions? Well metadata and I have had a somewhat schizophrenic relationship.

Back in the day (late 90's) I was in the same HP research laboratory as the [Jena semantic web](#) team. The problem, for me, was that I wouldn't/couldn't drink the cool aid - I kept my distance - forming an independent team to begin the ground work on ROC. This was not the easy path.

On reflection I now understand why I held out. I can now see, now that we've worked out the ROC abstraction, that there was still so much about the Web that was not understood - especially the progression from the single address space of the Web to the general contextual resource spaces of ROC. And, especially the practical consequences for the engineering of the solutions.



I'm interested in making things that work, and work really well. It felt like we needed to really understand the first order problem of engineering resource oriented systems - the second order metadata could wait...

...so lets just say the Jena team and I did not see eye to eye and leave it at that.

Maybe this history left me with some mental scars, but I have a certain wariness about putting the cart of metadata before the horse of systems engineering.

But, as Florence and her machine would say, [the dog days are over](#). Time to face up to metadata and coolly stare each other in the eye.

I think that we are now on very solid ground with ROC. Which allows us to fully consider metadata from the perspective of the general ROC abstraction.

I know that we have many metadata patterns in the ROC domain that are yielding daily practical benefits but I think that a clear articulation will help drive even more possibilities. I also believe that a general ROC perspective on metadata can offer feedback into the special case of the single address space case of the World Wide Web. So here goes...

How Metadata is Usually Described

We probably all know that Metadata is data about data. Look at, for example, the [wikipedia entry](#)...

Here's another perspective, from NISO, with the friendly sounding title "[Understanding Metadata](#)"...

The opening paragraphs of which go like this...

“What is Metadata?”

Metadata is structured information that describes, explains, locates, or otherwise makes it easier to retrieve, use, or manage an information resource. Metadata is often called data about data or information about information.

The term metadata is used differently in different communities. Some use it to refer to machine understandable information, while others use it only for records that describe electronic resources. In the library environment, metadata is commonly used for any formal scheme of resource description, applying to any type of object, digital or non-digital. [abbreviated]

There are three main types of metadata:

1. Descriptive metadata describes a resource for purposes such as discovery and identification. It can include elements such as title, abstract, author, and keywords.
2. Structural metadata indicates how compound objects are put together, for example, how pages are ordered to form chapters.
3. Administrative metadata provides information to help manage a resource, such as when and how it was created, file type and other technical information”

But it has always worried me that there are some pretty vague concepts here which rapidly lead to cans of worms, with many perspectives and interpretations.

I think the problem has been, as far as I've ever seen, that there are implicit assumptions that are not articulated.

As I'll show, I think we can use our ROC perspective to get back to first principles and give ourselves some more solid ground to build on.

About "about"

The root cause of trouble in the standard definition of meta-data as "data about data" is that little word in the middle: "about".

About

About tells us nothing, and yet implies too much.

For example, the standard definition completely fails to address identity. If something is information "about" some other thing, does it implicitly mean that the meta-information has within it the identifier of the data which it is "about"?

Sometimes perhaps, but as we'll see, it certainly doesn't have to.

So while "data about data" is easy to say, its pretty lazy. It is context free and abrogates responsibility for the fundamentals of the relationship between information and identity.

However this is precisely what ROC provides as a well defined foundation. It is a working and highly efficient, practical system that provides the basis within which identity and context form the information system. So it feels like ROC ought to be able to provide a more solid statement of what metadata is.

I think we can do a lot better than "about".

Metadata as Resource

Lets take the ROC axioms, defined in the first section of our [Introduction to ROC whitepaper](#), as the basis for a definition of Metadata.

In summary, the principles of Resource-Oriented 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

Incidentally I noticed that this whitepaper is now getting a little long in the tooth. Its main points are all still valid and correct but it refers to NK3 and it needs to explain the detail behind axiom #3 of relativistic contextual spacial relationships embodied in NetKernel 4 - one day I'll get to it!*

A Resource Oriented Definition of Metadata

With the ROC axioms in mind, a general ROC definition of Metadata is...



Metadata is a resource contextually associated with a resource

The key here is "contextually associated with" - where "contextually" immediately states that a metadata resource furnishes the associated resource with context. But, as we shall explore in detail in a moment, it also applies to the contextual nature of the association. Recall that in ROC "contextual" elicits the rich interplay between identity and structured address spaces (plural).

But first lets see what simply changing "data" to "resource" does for us, by referring to the ROC axioms.

From **Axiom #1** we already know what a resource is. Its an abstract set of information. So metadata is an abstract set of information contextually associated with another abstract set of information.

From **Axiom #2** it follows that a metadata resource may have one or more identifiers.

From **Axiom #3** a metadata resource's identifier may be resolved in an information context to a physical representation. By which we mean that metadata may be requested and resolved.

From **Axiom #4** we know that it takes computation to make metadata concrete. As we'll see this computation cost might be pre-paid (ahead of time) or it may be deferred and computed just-in-time. But it is *always* there. (*This explains why adding metadata to a system always takes time/money - you've got to pay the piper*)

We'll take it as read that **Axiom #5** is a "good thing" - for the same reasons as for any resource - its share and share alike in ROC, so don't go modifying representation state in a concurrent system.

Transreption, defined by **Axiom #6**, is actually saying that there is no "one-true" representational form for any resource. It follows that there is no one representational form for metadata. *I'm pleased about this - since, as you saw with the first section, the classical definitions of metadata almost instantly descend into the minutia and proliferation of formats and then rapidly diverge from there.* But, with a Resource Oriented perspective, we are forced to step outside of the representational form and accept that it is (fundamentally) irrelevant - and as we'll see, there are bigger fish to fry.

Axiom #7 says that a metadata resource once computed may have its own resource identifier just as any other resource. We'll see that this allows us to formally understand the notions of implicit and approximate identity. (*Amongst other things, "approximate identity" is what we use when we search for resources*)

Axioms

I suppose with this definition we have just introduced a new axiom to ROC, and there then follows an immediate requirement for a further axiom...

8. Metadata is a resource contextually associated with a resource.
9. A resource may have any number of independent contextually associated metadata resources.

We'll see how #9 is needed later.

The Purpose of Metadata

Just as with the physical world and [magnetic monopoles](#), it is a pretty rare thing to use a "Metadatum" - ie an individual isolated metadata resource.

Usually, we encounter metadata collectively. For example, the index of a book or table of contents. The reason for this is that...

The value and utility of Metadata increases when it is aggregated.

A very practical everyday example of this is, we save time finding stuff we're interested in by looking in an index, and an index has more value the more entries it has (*imagine if the index of a book only had one entry*).

So how can we understand this from the ROC perspective?

Well if metadata is a resource, then it's a set. Therefore an aggregation of metadata is a superset: the set of all metadata-sets.

At any given moment, "stuff that is interesting to us" is also a resource, one consisting of a sub-set of the aggregate set.

By applying constraints and/or transformations we can partition the aggregate set into a sub-set which satisfies our criteria. And, since our definition requires that metadata has a "contextual association", then from the constrained subset we must be able to resolve the resource. Therefore the constrained sub-set offers a context from which we can interact with the primary resources.

That, in a very concise way, explains what a Google, Yahoo or a Bing internet search engine is. It is an aggregate metadata resource set with the ability for us to apply selective constraint in order to reveal the identities (links) to bits of the web that interest us.

But sub-setting the aggregate set is not confined to "search" applications. For example if we're an access control architecture layer, the "thing that is of interest to us" is the set of resources for which suitable access credentials have been supplied. For a unit test system, it is the set of resources expressing unit test declarations. etc etc

OK, for completeness, lets apply our formal ROC axioms:

An aggregate metadata set is also a resource (by Axiom #7). Constraining the set to a sub-set is a transformation requiring computation (Axiom #5) and the subset is another resource (Axiom #7).

From this formal view, we now very clearly understand why the phrase "one person's metadata is another person's data" is a truism. The sub-setted aggregate resource is also a resource.

But it doesn't have to stop there, if it too is a resource then it too may have its own contextually associated metadata resources ... and so it goes, ad infinitum...

Notice that this discussion treats metadata resources in the abstract - we have had no need to concern ourselves with Axiom #6 and representation form. (We'll get on to "contextual association" soon I promise).

What's Metadata Good For? And Other Dumb Questions

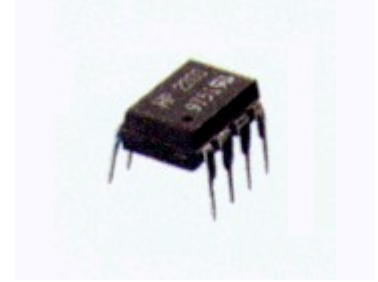
What's Metadata good for? Are there really 3-types of metadata, as categorised by NISO?

To the first: Errrr, anything which when aggregated has value in the context of your in-

formation solution. To the second: of course not, after the careful definition given above, its clear there are literally an infinite number of potential associated supersets and to classify them would be impossible. To see this just ask the rhetorical question "How many types of information resource are there?".

Comparative Example

One of my favourite analogies when thinking about metadata in software systems is to compare the [WSDL](#) specification of a SOAP web-service (take your pick they're all as depressingly bad as each other) with that of a mature metadata driven industry such as discrete electronic components.



Here, for example, is the National Semiconductor datasheet for its LM741 op-amp (the [op-amp](#) is the most common-or-garden component you can get and every manufacture offers a "741" device - its the "Hello World" of components)...

<http://www.national.com/ds/LM/LM741.pdf>

The 741 is a very simple and universal device and yet it still has a very detailed and rich specification sheet. Different engineering design decisions and requirements will dictate which subsets of this metadata are useful in any given context.

Notice its an aggregate resource - the composition of many different metadata resources.

Notice that it doesn't actually tell you what a 741 op-amp does, aside from a cursory memory-jogging "example circuit". You, as a trained electrical engineer are assumed to know not to connect OUTPUT to V+ in a positive feedback loop. The specsheet is pure metadata.

Notice something else, electrical engineering metadata is principally a human-readable resource. There will undoubtedly be a downloadable machine-readable SPICE simulation of the device somewhere - but the questions that matter in engineering are very often much more complex than the simple interface definition and electrical operating constraints of a CAD model (the equivalent of WSDL) - even for something as simple and ubiquitous as the humble 741.

The reason for this is that large scale systems always have more than one solution and require a delicate balance of compromises to be struck. (See my previous newsletter on why its called the "Art of Electronics"). Metadata helps the engineer find the best overall system balance to meet their given design brief.

(Oh how I wept when they came out with WSDL and yet they had the entire richness of the Web client-server browser model to provide rich human-readable engineering information about services. What does it do? How will it perform? When's it available? Where do I get credentials? Who do I call when it goes wrong? Such basic dumb questions need basic dumb answers... Oh! Web Services thow art tragedy. Tragedy!)

Metadata in Software Engineering

I won't attempt to make the category error of offering a general categorization! But in NetKernel we already use metadata resources to meet the following software engineering requirements...

- Specification
- Description
- Utility
- Functional Equivalence Comparison
- General Design Intent / Purpose
- Performance Expectation
- Logging
- Qualitative Comparison of Architectures
- Statistical Operational Characteristics
- Testing
- Structural Constraints
- semantic Constraints (small-s - see below)
- Contractual Operational Constraints
- Trust and Trust Boundaries (aka Security)
- Dynamic Configuration
- Operational management

I'm going to keep with the specifics of NetKernel for the next section - since as the first (only) general ROC system it concretely allows us to explore the ways in which "contextual association" can be understood.

However I hope it will be apparent that the patterns which are discussed are general and are applicable more generally in extended architectures.

Contextual Association

The unarticulated, but necessary, implication of "about" in the previous definitions of metadata is that the information in the metadata resource must "somehow" be related to some other information.



In ROC we understand that this must mean that the metadata resource either explicitly references an identity of the resource or, possibly, implicitly provides sufficient bread-crumbs to be able to reconstruct an identity.

We shall see that this is richer than simply direct or indirect use of a resource identifier, we are able to include the "locality" of the scope of the ROC spacial architecture as a factor in ensuring the *contextual association* is meaningful. We are also able to allow that a resource may have one or many identifiers (Axiom #2).

Here we'll consider and define the manner in which "contextual association" can be implemented and consider the significance of patterns using both explicit or implicit identity for the associated resource.

As an example, notice in the 741 op-amp [datasheet](#) the liberal scattering of **Order Number LM741H** etc - these are explicit identifiers of the part number to put on your purchase order which will "reify a representation".

So lets look at the possible ways of enabling the *contextual association of metadata* resources, and lets take them in-order of "relative locality", most localized first...

Attributes

A completely localized approach to associating metadata with a resource is to "decorate" the primary resource with attributes. That is, the resource itself contains embedded metadata which can be independently parsed and extracted from it by an aggregator.

A key constraint of this model is that metadata attributes *must* be "out-of-band"; that is, the primary information of the resource is unperturbed by the addition of the metadata.

The introduction in Java 5 of [Annotations](#) is a localized attribute-based metadata model.

Equally, the [RDFa W3C standard](#) enables RDF metadata to be expressed as attributes embedded within an HTML/XHTML resource.

In both cases it is not necessary for the creator of the metadata to explicitly state the identity of the resource to which the metadata is associated. The localized context of the attribute is sufficient for an aggregator to infer the identity of the resource in question, and even, for example in the case of RDFa, metadata associated with the identity of another referenced resource.

The advantage of the attribute model is that it is relatively simple to augment a resource with metadata.

However, a considerable draw back, if you accept the premise of Axiom #9 (introduced above), is that our general ROC expectation is that when metadata is a resource then inevitably any given resource will be associated with more than one metadata resource. Therefore cramming different attribute models into one resource would mean that things will get messy very quickly.

A further drawback is that an attribute based model can only work with a strict representation model for the resource. It cannot cope with Axiom #6 whereby a resource is abstract and representations may take unlimited form. Transreption does not (cannot) promise to isomorphically preserve attribute metadata.

However, as we see with RDFa and HTML resources - these trade trade-offs have been deemed acceptable, and the HTML resource representation so common, that the limitations of the approach are a price worth paying for the anticipated benefit of having a simple model for web content to be augmented with metadata resources.

NetKernel - *NetKernel does not implement an attribute based metadata model in the standard modules' endpoint declarations. However the standard and active grammar specifications allow attribute metadata to describe the nature of arguments, and this shows up in the boilerplate documentation for every endpoint. But, just as with the Web, you may readily adopt the attribute model within any resource representation you produce in a NetKernel solution.*

Response Associated

Any resource provider may supply arbitrary and unlimited metadata as secondary state conveyed in parallel with the response to a request for the resource. The most obvious example of this approach is mimetypes (both in HTTP and email systems), another is the SOAP message header.

By definition, since you must request it before you even access the metadata, the identity with which the metadata resource is associated is known and so may be explicitly incorporated with any metadata aggregation.

The drawback of this model is that the provider of the resource must be tightly bound to the metadata and therefore must be reimplemented if new or additional metadata resources are to be associated - again causing friction with our expectations from Axiom #9.

Furthermore the metadata resource is "out-of-band" to the resource oriented domain. It follows that an aggregator must, ahead of time, be able to discover and request (compute, Axiom #4) every resource in every address space in an information system in order to present any accumulated aggregate metadata driven services.

As with attributes this model is not guaranteed to survive axiom #6.

NetKernel - you can easily implement this model in a NetKernel solution. The INKFRresponse object provides a convenience method to specify mimetype, but more generally it allows arbitrary response headers to be specified on a response (mimetype is also just a header, it only has a setter method for convenience, since HTTP applications almost always need the mimetype to be specified). Attached metadata headers would use a common convention to name the header so that it is detected by an aggregator. Furthermore the mapper allows arbitrary response metadata to be declaratively defined and associated with any request to a mapped logical endpoint - so unlike with SOAP, you can externally decorate a resource set's representations with metadata at runtime.

Parameterised META Resources

An endpoint in an ROC system should respond to a META request. The response is a set of parameterized metadata resources specific to that endpoint. Typically the META request is issued for the identifier of the endpoint - not an identifier of a member of the resource set managed by the endpoint. As such it is a direct request for the metadata about the endpoint and not about the endpoint's resources. Although clearly often there is a close relationship between the two.

Here for example is how to request the META resource of the Groovy language runtime (id: GroovyRuntime) using the request trace tool...

The screenshot shows the NetKernel Request Resolution Trace Tool interface. At the top, there's a navigation bar with links: Home, Status, Control Panel, Developer, Explore, Documentation, Apposite, Tweeter, Comserv, WINK, nk4um. Below this is a search bar and a 'Up 5 days' indicator. The main section is titled 'Request Resolution Trace Tool' with the subtitle 'Inject resolution requests to test address-spaces'. It contains several input fields: 'Module' (set to 'urn:org:netkernel:lang:groovy - 1.7.1'), 'Space' (set to 'Lang / Groovy'), 'Identifier' (set to 'GroovyRuntime'), and 'Verb' (set to 'META'). There are three buttons: 'Resolve', 'Inject', and 'Display Representation'. Below these fields, a table displays the response details:

Representation Type:	NK Metadata (org.netkernel.layer0.meta.impl.EndpointMetaImpl)
Representation toString():	org.netkernel.layer0.meta.impl.EndpointMetaImpl@27ecfcd9
Expiry Determinants:	NeverExpired
IsExpired?:	false
Cost:	25
Response Fields:	none

Which results in the following metadata resource (transrepted to an XML serialized

HDS structure)...

```

<meta>
  <name>Groovy</name>
  <desc>Groovy language runtime</desc>
  <icon>res:/org/netkernel/lang/groovy/img/icon.png</icon>
  <fields />
  <class>org.netkernel.layer0.meta.IEndpointMeta</class>
  <identifier>GroovyRuntime</identifier>
  <interface>
    <verbs>
      <verb>SOURCE</verb>
      <verb>SINK</verb>
      <verb>EXISTS</verb>
      <verb>DELETE</verb>
      <verb>NEW</verb>
    </verbs>
  <grammar>
    <group>
      <group name="scheme">active</group>:
      <group name="activeType">groovy</group>
      <interleave>
        <group>+operator@
          <group name="operator">
            <regex type="active-escaped-uri-loose" />
          </group>
        </group>
        <group min="0" max="*">+
          <group name="argName">
            <regex type="nmtoken" />
          </group>@
          <group name="argValue">
            <regex type="active-escaped-uri-loose" />
          </group>
        </group>
      </interleave>
    </group>
  </grammar>
  <args>
    <arg>
      <name>operator</name>
      <desc>Groovy program to be executed</desc>
      <fields />
      <type>REPRESENTATION</type>
      <reps>
        <rep>
          <hash>VP/P/GroovyCompiledRepresentation</hash>

```



```

        <name>CompiledGroovyRep</name>
        <class>org.netkernel.lang.groovy.representation.CompiledGroovyRep</c
lass>
        </rep>
    </reps>
</arg>
<arg>
    <name>varargs</name>
    <fields />
    <type>REPRESENTATION</type>
    <reps>
        <rep>
            <class>java.lang.Object</class>
        </rep>
    </reps>
    <default>varargs</default>
</arg>
</args>
<exs />
<reps />
</interface>
</meta>

```

For NetKernel and endpoints in a Standard Module, this metadata resource consists of intrinsic automatically generated metadata derived from the implementation (for example the verbs it handles). Other metadata is extrinsic and is declaratively specified as parameters at the point in the address space where the endpoint is instantiated - for example the name, desc, *grammar* resources.

The metadata structure of the standard module endpoint is internally held as [HDS](#) and as such is extensible with user specified fields. Currently it is only possible to programmatically extend the metadata model and it is not a documented feature since the META resource is used by many system tools. However we are currently undertaking the work to allow an extrinsic user-supplied open-ended metadata parameter to be declared. We expect this will be in the form of HDS which is able to be a list, tree or forest structure and so will allow arbitrary metadata formats to be combined at the point of declaration (eg, RDF, XML, Domain-Specific etc etc).

The endpoint identifiers of all endpoints in an ROC system act as a primary key and must be unique within any given address space. The `active:moduleList` endpoint in `ext:system` provides an aggregate metadata resource showing all public address spaces and the identifier of each endpoint in the space. With this breadcrumb "contextual association" it is possible to find and issue META requests to any endpoint anywhere in the system. Many of NetKernel's system tools use this resource oriented metadata as the basis for their opera-

tion.

It follows that this, and perhaps some simplified user-centric accessors, will enable very elegant aggregation patterns of application-specific user-specified metadata resources. We anticipate that this will provide significant opportunities and go a long way beyond the capabilities of low-level programming language metadata models, such as Java annotations, to allow general metadata driven application architectures (as [recently demonstrated](#) by Brian Sletten without even having the upcoming capabilities).

Currently META is defined in ROC and implemented in NetKernel, but it does not have an analogue in the World Wide Web - however, it would be relatively straight forward to add this verb to the HTTP methods.

As with the previous methods, the Parameterized META approach is localized. Which makes it convenient, but it also has a little friction with respect to Axiom #9 since its metadata is tightly coupled. Even so, it still satisfies Axiom #9 since more than one form of metadata resource can be simultaneously associated with the endpoint. Also unlike before, it satisfies Axiom #6 since this is a true resource representation and is requested in the ROC domain and so can be transrepted/cached etc.

*A draw back of this method is that the metadata resource is for the endpoint and not one of the resources that it may be the accessor for. As mentioned, there is often a close correlation, but it means that the metadata supplied is for the **entire set** of resources it manages. Which is great for access control, non-functional architectural configuration, logging etc etc but falls short of the even more general approach offered by other patterns we'll see in a moment. That said, it still significantly exceeds the capabilities of Annotations.*

Extrinsic Metadata Resource

The most general model for contextual association of metadata is to acknowledge that metadata is a first class resource in its own right with an identity (By Axiom #1). A true metadata resource must exist in an address space (By Axiom #2). The information in the metadata resource may be requested and reified to a concrete representation (Axiom #3). It may be static (pre-defined) or dynamically computed (Axiom #4). It may be transrepted (Axiom #6). It may be composed, aggregated and transformed to form other resources (Axiom #7).

So far, so like any other resource.

We do not care what the metadata expresses, but it must be *contextually* associated. But, since it lives in an address space it cannot, as we've seen in the previous models, be locally associated. Therefore it **must** contain a direct or indirect reference (identity, link) to the resource with which it is associated.

In certain communities you may hear the term "linked data". We won't adopt this term here since we're trying to be as general as we can and it has some pre-existing connotations especially, as we'll see, a built-in mono-spacial assumption.

Finally, since the metadata resource is entirely independent from the contextually associated resource it follows that this pattern fully satisfies Axiom #9. Any resource may simultaneously be associated with arbitrary numbers of metadata resources.

Mono-Spatial Systems: Semantic Web



The use of RDF structures located in the World Wide Web is an example of this model in a single address space (the Web). An RDF resource may be located in the Web address space with a URL. The resource(s) it is associated with are explicitly referenced in the RDF graph. Sometimes relative URIs may be used where the identity of the metadata resource provides the context to resolve these to an absolute identifier.

RDF is a graph of triples and it implicitly assumes that there is only a single address space in which all resources are located (the Web). Therefore it may reference other RDF metadata resources in the same space and so introduce secondary contextual associations for the resource. Forming a distributed web (superset) of metadata.

General multi-spacial ROC systems

The *extrinsic metadata resource* pattern is the most common metadata model used within the NetKernel ROC system.

It is seen employed in single address spaces - for example it is the way that general configuration metadata is automatically discovered for the relational database tools - whereby they request a "standard resource" identifier `res:/etc/RDBMSConfig.xml`.

However its power is most readily seen when we consider that a NetKernel ROC system consists of multiple independent address spaces. (*Imagine that there were unlimited parallel World Wide Webs of arbitrary scale and you'll have a glimpse of general ROC.*) Such a system must have metadata driven applications allowing aggregate metadata resources to be used to provide operation, configuration, description etc etc across and over the system.

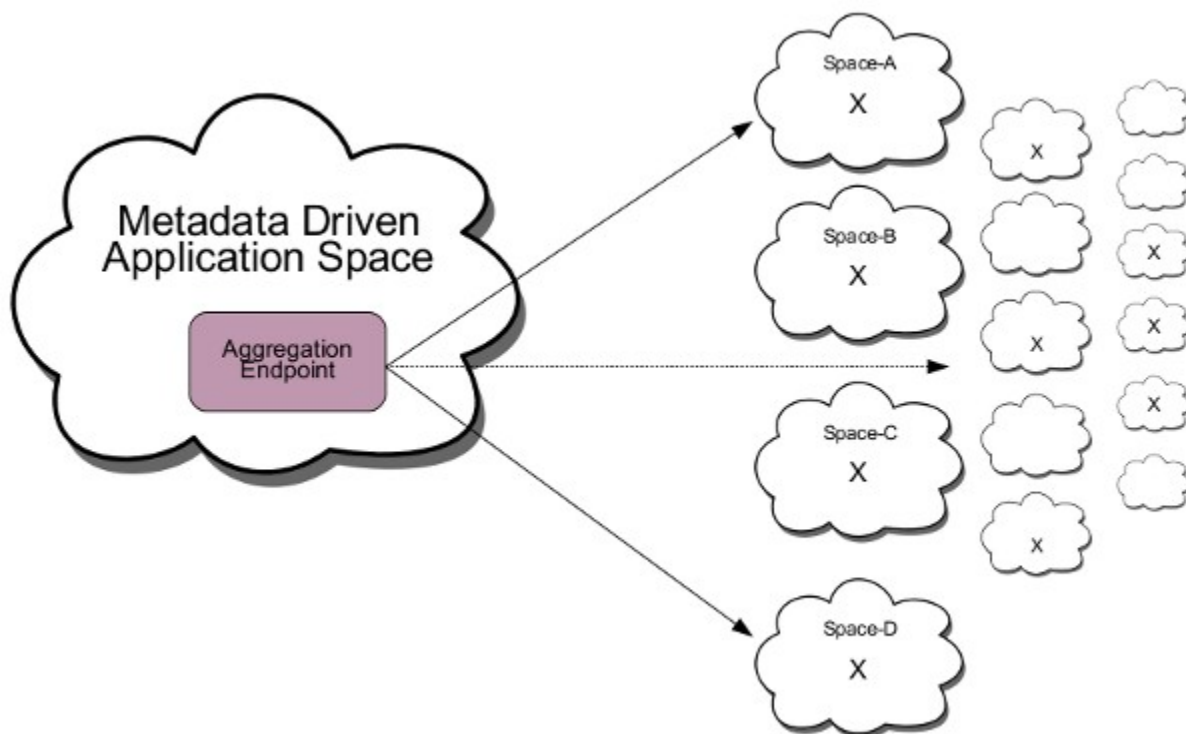
The best way to describe this model in action is to consider the specifics of the "res:/etc/X" pattern. For example consider the NetKernel XUnit test system, the documentation application, the search system, the space explorer, the nCoDE runtime development environment etc etc.

The diagram below shows a metadata driven application which requires an aggregation of metadata resources resident in arbitrary independent spaces.

The essence of the pattern is that we may present suitable metadata to such applications by implementing a resource in the address space which has a common name in each space (at least those spaces wishing to share this metadata).

For the NetKernel tools we have adopted a convention that their metadata resources will use identifiers in the **res:/etc/system/*** resource set. Hence documentation is discovered at **res:/etc/system/Docs.xml**, unit tests at **res:/etc/system/Tests.xml** etc etc.

A tool requiring an aggregation of the metadata resources must interrogate all the spaces to discover if they provide a **res:/etc/system/X** resource. This is relatively simple to achieve since the kernel provides a resource which is the set of all (public) address spaces. All that is required is to use this list and to knock on the door of each space and request the metadata resource (shown in the diagram, where for clarity we've just called the metadata resource "X").



Each metadata resource X has within it a direct or implicit reference to the resource with which it is contextually associated. So for example each `<doc>` entry has a reference to the documentation resource which it describes, equally for unit tests etc etc. Importantly the

identifier need not be globally unique - it is sufficient that it is contextually relevant from the point of reference of the metadata resource in the address space.

However, it is beholden on the aggregator that it should maintain a record of the spacial context for which the metadata resource was aggregated - so that, by breadcrumbs, the relevant related primary resource can be disambiguated should the metadata be of interest.

Once the aggregator has obtained the super-set of metadata resources it is free to use this as a resource within its application domain. For example, to show a table of contents of documentation, to request and render a composite view of a document, to show and provide execution of a unit tests etc etc.

As we've said, metadata is frequently used in order to find "resources of interest" - it follows that usually a metadata driven application will subsequently actually want to access the primary resource which is contextually associated via the metadata.

It would be quite possible, with the knowledge of the space and resource identifier, to construct a request for that specific primary resource in the application. However this use case is common and so there is a tool to do the work for you. We call this type of request the "wormhole pattern" and we provide the [active:wormhole](#) tool in layer1.

Aggregator Implementation

It is relatively straightforward to implement an aggregator endpoint to implement this pattern. However, since it is so common, we provide the [active:SpaceAggregateHDS](#) tool in layer1- which is specific to the HDS resource model and which requests a named resource from every space in the system to create an aggregate HDS metadata resource.

HDS

We use HDS as our standard internal resource model for metadata for the following reasons:

- The [HDS](#) structure is a flexible name-value node which may have one parent and multiple children.
- Its flexibility allows it to support *list*, *tree* and *forest* structures.
- It supports arbitrary comparator filters to apply efficient sub-setting operations.
- It may be *inverted* to provide a value-keyed map for "indexed operations".

In short, with a simple and very computationally efficient data structure it provides all the flexibility we have needed to operate the NetKernel system.

One of the singular benefits of HDS is that an aggregate resource may be rapidly sub-

setted by using XPath expressions. Path expressions are easily comprehensible by human beings and they are essentially a continuation of the resource address space - since they are a resource identifier to the subset of a resource, resolved within that resource. XPath expressions on an HDS structure result in iterable node lists which are very simple to process and can be easily presented as resources to either synchronous or asynchronous sub-processes.

Of course, as I've said, axiomatically, the representation form of metadata resource is irrelevant. So if you have a specific need for another representational resource model then just look at the source code for the SpaceAggregateHDS tool and implement your own aggregator - its dead simple.

Power of the Extrinsic Metadata Resource model

To see the full power of the general extrinsic resource model we should consider the formal ROC foundations again.

Since these metadata resources are in the address space they may be transrepted (Axiom #6) - which gives innate decoupling between the metadata provider and the metadata driven applications.

However the most powerful point is to understand that any resource may be dynamically computed (Axiom #4). It follows that the NetKernel system tools are completely open to dynamically computed extensions (for example, the dynamically generated boilerplate for an endpoint's documentation, the dynamic reconfiguration of the nCoDE palettes etc etc).

Furthermore all metadata resources and composite aggregates (Axiom #7) are cacheable with associated dependency consistency, as with any other ROC application. Therefore metadata driven ROC applications exhibit the same scaling and computational efficiencies as exhibited by any other ROC solution.

Finally, there is a semi-magical consequence of this pattern. It provides the basis for the "dynamic import" pattern, in which the spacial architecture of an entire ROC solution is dynamically structured based upon aggregate metadata. **NetKernel supports metadata driven architecture.**

Its not all roses though, as ever there is an engineering balance to consider, having an extrinsic relationship with independent metadata resources is more flexible but the trade-off is convenience. Its generally less hassle at development time to bind metadata tightly at the point of origin (as shown in the locality factor of the other methods). However inconvenience is a one off cost, whereas metadata has value for a long time. You may recognise that the [extensible meta parameter](#) approach we are proposing is a half-way-house compromise that will be perfectly suitable for certain patterns and still be pretty convenient.

Engineering is an art form. Its all about finding an elegant balance.

Summary

Hopefully this discussion has demonstrated how the wider-perspective of Resource Oriented Computing allows us to uniformly treat metadata as just another resource. Everything is a resource.

There are still many things I could talk about. Like for example, I slipped in the term "aggregation" and then didn't explain it. Just off the top of my head we could go into aggregation with respect to...

- metadata as extracted sub-set of the resource
- metadata as side-effect state (logging)
- metadata as measurement of operations
- metadata as creative artefact

I've also stepped back from a full discussion of the semantic web. You can be sure I have opinions and ideas here - but I prefer to speak when I've got concrete solid examples to back-up what I'm talking about. So that'll have to wait for another time.

I'll close with this thought. Back at HP in days of yore, John Erickson used to bang the table (literally) and, eye's popping and veins extended, scream: "Metadata is the lifeblood of e-commerce". Well "Jahn", I'll steal your thunder...

Metadata is the lifeblood of ROC!

and, maybe, just maybe, I've convinced you that...

ROC is the foundation of metadata

Corollary: The Search Giant's versus RDFa

Incidentally, the discussion above may help offer perspective on an interesting skirmish over the future of the Web. You may know that the leading web search engines (Yahoo, Bing and Google - though Yahoo uses Bing so there are only two parties really) have recently published and promoted a microdata specification for attribute metadata in HTML resources...

<http://schema.org>

Even though they claim that they will still aggregate [RDFa](#) it is actually a pretty strong strategic play.

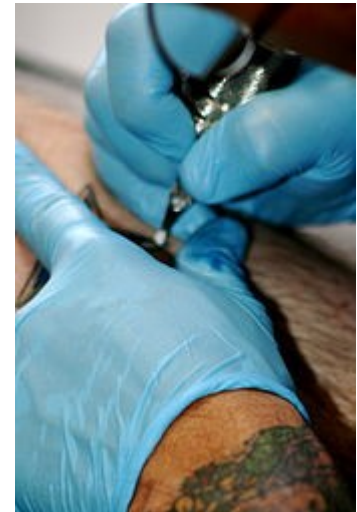
Why so? Well they say that their format is extensible and that "given adoption" they will recognise commonly adopted extensions, but that means that only metadata resources which are "blessed" are allowed into the walled garden.

You would hope this happens fairly quickly, a cursory glance at the recognised types in the [full type list](#) shows how laughably primitive the implementers of the format have been. I'll not even get into the innate cultural imperialism. Suffice to say that TattooParlor owners will be happy - but that [TattooParlour](#) owners are plumb out of luck.

Hopefully, what I've shown is that "contextual association" is the essence by which a resource is said to be metadata. But by having their own blessed format, it ensures that the search giants are able to build metadata driven applications that are specifically tuned for this. In essence it allows them to guarantee and maintain control of the "context" by which you see the Web's resources.

This is reinforced when you see that this is a centralised model and it guarantees and reinforces the role of the aggregator as mediator to information. Search algorithms are not open to inspection, and this specification offers no warranty that your page rank will be undamaged should you choose not to use their markup.

But then what do you expect, being the central aggregator and tuning the sub-setting of aggregate metadata resources is their business model.



Anyway, ultimately and fundamentally the format of metadata is a dull and uninteresting thing. Its your choice. But if it were my data and I wanted to describe it - I'd hope that using open extensible formats would be a better long term decision. ♠

On State and Consistency

Peter Rodgers, PhD

It's true to say that the off-the-shelf component libraries we provide with NK are stateless. Therefore when assembling a composition (a process, a pipeline) the overall flow through tools is transient - state starts on the boundary of the system flows through, stuff gets done, responses return and the endpoints resume their initial condition.

That's the general picture. But of course as we run the system we accumulate representational state at various natural boundaries in the solution. In NetKernel all resource requests are potentially cacheable (in comp-sci terms, NK has systemmic memoization). Quite naturally representational state percolates upwards towards the client-side.

In most cases an application requests external state from persistence mechanisms (DB's etc). Importantly however, any given application is usually about building composite resources from the "atoms" of this external "normalized state". The Golden Thread pattern allows externally sourced state and the derivative composite state higher in the upper parts of your application to be kept consistent as state changes occur.

If you've not heard of the **GoldenThread pattern** then there's a demo you can install `demo-goldenthread` with `Apposite`. It includes embedded video presentations that discuss the details and which you can view [here \(part1\)](#) and [here \(part2\)](#).

You could debate that one definition of an "application" is to provide a contextual relation between one set of "denormalized" state (the client-side surfaces) and one (or more) sets of "normalized" state (business state). Security is then the constraint of the denormalized state such that the application channel can only reach that subset of the normal state that is deemed "suitable". This is why in the 3C's of ROC, security/validation are part of the 3rd C: Constraint, which can be applied retrospectively and orthogonally to an application architecture.

Returning to the general picture. As I said, while the off-the-shelf services we provide are stateless, there is actually nothing in the abstraction which mandates stateless endpoints. It's just that in compositional architectures statelessness means that systems remain consistent without a developer/architect needing to give it any thought.

However, imagine we have a Geiger counter endpoint: `active:geigerCounter` - which records external gamma ray events. This endpoint would be stateful. Each request to this endpoint would return the current event count.

But how do we enforce consistency now? Well we ensure that any representation that we return is only cacheable for as long as the internal count state is unchanged. We can associate a dependency to all representations that expires as soon as the state changes (we can do this with an expiry function on the response). `active:geigerCounter` is stateful but it is implemented to ensure system consistency.

Now lets think about a random number generator: `active:random` - this service is stateful in the sense that it has a random number generator whose operating state guarantees that it will produce random numbers and is not going to return the initial seed value on each request. For system consistency we would make this tool return a response that was always expired. Therefore no matter how this random resource gets composed up into composite state above, the dependent state will be "on a branch" that can never be reached by anyone else ever again.

We can play games with the engineering balance of the state in the system too.

We can introduce approximate consistency. For example we can use time as a consistency approximation. We can say things like, this resource is valid between $t=0$ and $t=x$ and don't come back to me before x . For example, with the Geiger endpoint we could decide that its good enough that we know the count to within a tolerance of 10 seconds. We'd do this by setting the expiry on the `INKFResponse`...

```
response.setExpiry(xxxxxx, T)
```

Where `xxxxxx` is one of...

<code>EXPIRY_ALWAYS</code>	This is always expired and anything derived from it (dependent on it) is expired
<code>EXPIRY_CONSTANT</code>	This is valid until time <code>T</code>
<code>EXPIRY_DEPENDENT</code>	This is valid as long as every resource we depend on is also valid (default)
<code>EXPIRY_FUNCTION</code>	Before serving from cache, invoke this function to determine validity
<code>EXPIRY_MIN_CONSTANT_DEPENDENT</code>	This remains valid until time <code>T</code> but will expire earlier if a dependency expires
<code>EXPIRY_MAX_CONSTANT_DEPENDENT</code>	This remains valid until time <code>T</code> or until a dependency expires, whichever is the longer

EXPIRY_MIN_FUNCTION_DEPENDENT	This remains valid as long as function is valid but will expire earlier if a dependency expires
EXPIRY_NEVER	This is valid forever

So, by playing with time you can introduce [Nyquist](#) sampling criterion to your application's architecture.

We can also set our own expiry functions too. For example, its likely that a timed expiry on a Geiger counter is a bit foolish (especially if you're monitoring a nuclear power plant) instead it would make more sense to have an expiry function that expires every 100 hits - no timing necessary!

I'd classify this sort of pattern as "windowing" state expiration, we're sub-sampling the rapidly moving state to give transient consistency over a window of stability. Again, this is where Nyquist criterion become part of your architectural armoury.

One thing to make clear is that a user-defined expiry function does not require that you implement an internal thread to monitor an endpoint's internal state and manage consistency. An expiry function is only invoked if a given representation is in the cache and could possibly satisfy a request. In this case, the thread of the requestor that wants the resource, is used to call into the endpoint expiry function. In fact this is a general principle, state consistency is not pre-emptively managed, only if/when a resource is requested is the consistency of any possible pre-computed representational state determined ("just-in-time consistency").

In some instances we might actually want to have direct accesss to the rapidly changing stateful endpoints, like the raw Geiger counter endpoint. But nothing prevents us introducing wrapped smoothing services, `active:windowingGeigierCounter`, which would apply the windowing expiry over the rapidly moving source. We can build up assemblies of faster and slower changing statefulness. Of course, you see that we can do the same against external data sources too.

By now, you'll recognize that nothing in the abstraction prevents you writing an endpoint that is more like a full stateful client (eg like a browser, an orchestration workflow tool). A simple example in the off-the-shelf tools, is the Cron transport - this allows you to register a "space-time request" which is held statefully in the cron transport until at the given point in time your request is issued to the given space.

Big Picture

Hopefully its clear from the discussion that with ROC you have the freedom to introduce engineering tolerances in order manage "good-enough" system state. For any given application, there is always some critereon below which a fair approximation is good enough to maintain consistency with reality.

Why does this matter? Because data and reality are not the same thing. Data is always an approximation and being able to make intelligent choices about these approximations leads to high quality, robust, tolerant, adaptable and efficient (fast) information systems.

You are one such system. Cinema and TV are possible only because your image processing system is based on an approximately consistent representation of the photons arriving at your eyeball. Your Nyquest sampling rate is about 25Hz. That's good enough to detect a lion starting to run towards you - but its too slow to see the shockwave of a bullet or the individual frames of Avatar.

With an ROC solution, accumulated operating representational state can have approximate fidelity with reality. But *critically*, **within** an ROC solution a requestor is given a **guarantee** that in a given context they may issue a request and receive state and that state will be **systemmically consistent**.

System state consistency is assured irrespective of if the system has just booted or has started to find cacheable information, or the state is "moving" within a given resource set. The consistency is contextual - its not the case that the system has to be globally consistent, only consistent such that for a given spacial context two equivalent requests will receive the same resource state (all other things being equal).

The ROC abstraction embodies an internally consistent contextual computational state space - which, for any given application, naturally discovers the minimal transient state (min E, energy) and, through transreption, the most useful form (min S , entropy).

[Entropy](#) is a really important concept. When you get a feel for it, NK and ROC gives you an environment in which you can engineer it. You maybe never thought about it but Transreption, the isomorphic restructuring of one representation state to another, is actually formally *entropy transformation*. Information is conserved, entropy is reduced. As with any other request, transrepting entropy reductions are cached, so **NetKernel discovers the most valuable state in the most valuable structural form (lowest entropy)**.

There's lots of hype in the market, but for truly green computing you must start with the thermodynamics of your information. **ROC enables green software.**⁷ ♠



⁷ Amory Lovins, who created the Negawatt movement, and his team at the [Rocky Mountain Institute](#) specified [Design Recommendations for High-Performance Data Centers](#). “Eliminate “bloatware” and make code that allows chips to scale up and down. Bloatware refers to the drive to make continually larger, fancier applications that results in users’ continually upgrading equipment to run those applications”. In other words reduce energy consumption by making your processors work less. Seems to be perfectly in line with ROC, isn't it?

DPML – Declarative Markup Language

The Declarative-Request Process Markup Language (DPML), is a simple, lightweight language. With DPML you can write programs that directly request information resources and leverage various resource models without knowledge of low-level APIs and objects.

Although DPML is a simple language, it leverages the ROC abstraction which allows DPML programs to implement sophisticated applications. Many developers find DPML a valuable tool because it allows them to isolate the definition of process flow at the logical level from details that are properly kept within the physical level.

The syntax evaluated by the DPML language runtime engine is based on an abstract syntax tree (AST). This makes DPML independent of any particular text-based syntax. The current DPML implementation uses an XML based syntax which is compiled (transrepted) to the AST syntax. Being an XML syntax it is well matched to patterns that require dynamically generated processes.

A fundamental difference between DPML and other languages is that DPML works only with requests and responses; it does not manipulate nor even access the representations returned by endpoints within their responses. This clean separation between the logical and physical levels matches the same clean separation between the logical and physical levels in the ROC abstraction.

In addition to processing requests and responses, this implementation of DPML includes several plug-in capabilities such as sequence, if-then-else and exception handling. ♠

DPML – A very unusual language

Peter Rodgers, PhD

“DPML is a very easy to learn language”

We released an update to the **lang-dpml** package to fix a bug in which the `<switch>` conditional was not correctly validating. This provided the context for today's short story and presents the opportunity to discuss why DPML is a very unusual language.

On face value DPML appears to have within it support for **if**, **switch**, **log**, **throw**, etc etc. The sorts of conditional logic and process control you'd expect in any language. But the truth is that DPML does not have any in-built functions at all. Nor indeed does it have any in-built types. Not even "int".

When you use DPML what you're doing is co-ordinating a series of resource requests. DPML's only "type" is the [declarative request](#). When you write an `<if>` statement what actually happens is that the tag gets converted to a request to the [active:if](#) endpoint - provided as a service in the DPML language's rootspace.

If you want to extend the DPML language you can create your own endpoint and bind into the language with a [tag declaration](#). The metadata of the target endpoint is used to construct a request based upon the tag used in your DPML code.

Now, I'm sure language scholars amongst you will be able to point out other examples of extensible languages. Not least in variants of LISP.

The next part is something I don't think any other language can claim. For, did you know that DPML is a "stateless language"? Well not quite, it contains just one single item of state - a "program counter" which maintains a reference to the location in the code which is currently being evaluated. But DPML holds no variable state!

"Hold on hold on! But you can do assignments to named variables!!"

Yes you can. But whenever any representation state is returned in a request it is assigned to that named assignment (variable) but in fact it lives inside an address space which is contextually scoped to the current execution. **DPML variables are resources that live in ROC spaces.**

If you've followed the discussion about [Pass-by-Value spaces](#) - you'll know we discovered an elegant way to turn push into pull. DPML simply takes this to the ultimate level. It says "all state is extrinsic" the execution state is a set of resources - all functions in the language will pull this state.

So now you see that if this is the underlying basis of DPML, it makes complete sense that all the language's functions are extrinsic too.

The really clever part about DPML's implementation is that it doesn't just construct one state bearing space. It actually uses the same approach to provide state to the execution of closures. If you think too hard about it you start to understand that during runtime, the DPML code explodes into a dynamic sea of state spaces.

Why the heck would you go to all this trouble? Well its just as we always say - when you make state extrinsic there's the opportunity that someone else can reuse it. What if you have an active:if request and its condition has not changed since the last time it was requested - the "true" representation can come straight out of cache...

There is no faster answer to a computational algorithm than not to compute at all.

Incidentally - **DPML is a very easy to learn language** - when you've learned the mapper - then DPML is just sequencing requests. But if you don't want to learn DPML you can always use nCoDE. Since nCoDE actually gets transcribed to DPML and runs on the DPML runtime! ♠

nCoDE

nCoDE [en-kohd] -

1. NetKernel Compositional Development Environment
2. to convert (a message, information, etc.) into code.

The **nCoDE** Module provides a Compositional Development Environment (CDE) paired with a Language Runtime. The CDE is an web-based AJAX visual editor for editing the programs.

The word *compositional* refers specifically to the second C of the 3C's Construct/Com-pose/Constrain - an ROC development methodology.

Composition is the process of assembling pre-built endpoints into systems and applications. Usually this consists of two distinct parts:

- 1) definining and structuring spaces,
- 2) orchestrating composite processes within those spaces. nCoDE helps with the latter of the two.

nCoDE leverages the capabilities of DPML (Declarative Process Markup Language) by trans-lating it's internal definition into standard DPML code for execution. DPML takes a very pure approach to composition - it does nothing more than orchestrate requests between endpoints and the definition of literal representations. DPML supports both an imperative and a functional approach but nCoDE uses only the functional approach. Though the imperative approach can be modelled using the built-in sequence endpoint.

Most "drag and drop" visual programming languages use the dataflow programming paradigm. However nCoDE, in common with Yahoo! Pipes, uses a functional approach. Both approaches look similar in their visual representation, both utilise "black box" components. Simple pro-grams may actually look identical in both approaches. They differ significantly in how they execute.

Dataflow programming uses a model where each component can output changes and these changes then ripple through network as needed. The most well known example is a spread-sheet. In dataflow the data is pushed.

Functional programs are driven by their response and are in contrast pull based. Each function is only invoked if it is specified as an argument to a function that needs to be evaluated. This happens recursively and usually lazily. Functional programs have a deep foundation as expressed in lambda calculus. Functional programming is very closely aligned with ROC the main differences being:

- ROC externalises scope into programmatically controllable address spaces.
- ROC blurs the boundary between pure and non-pure functions with rich validity semantics on responses. ♠

Separating Architecture from Code

Tony Butterfield

The [MP3](#) format is a great technology for compressing audio because it can be progressively configured to reduce the complexity in an audio stream in the parts that humans are less able to perceive. It understands the psycho-acoustics of human hearing and the typical waveforms that constitute music. By working in a constrained domain it can represent the data in a more concentrated form. Similarly JPEG is a great technology for compressing photographs because it can reduce the complexity in parts of an image that humans are less able to perceive. It knows that typical photographs contain large areas of flat colour or slowly changing gradients; it knows that our eyes see less colour information than they see luminance. The algorithms behind these technologies are very smart, they understand the nature of the data and take advantage of that fact to represent the information in a more concentrated form. There are many more examples of these domain specific compressions in video encoding.

At first sight [ZIP](#) is great technology for compressing general purpose data streams. It can produce a representation of the original data which is smaller and without loss. The ZIP algorithm works by finding common patterns throughout the data and uses a language to express those patterns. Whilst this works for "typical" data it doesn't always work. By "typical" data I mean real world data than contains patterns and is not completely random. Hold that thought...

Now lets switch gears to think about software. Imagine a language that could express typical software patterns and architectures in a more compact way - it would translate to less code. If we understood what constituted the essential essence of typical software systems with a language to express it then we could "decompress" this to create an instantiation of that system in all it's gory detail needed by the complex hardware of today's computers. By this I do not mean just putting some source code through a compression algorithm. That would be no good - because the compressed data could not be authored. What I mean is a language for expressing the system which enables you to achieve a lot with a little.

One example of what I mean is a technology like [Ruby on Rails](#), a fairly small set of files can create a rich application. A small set of input data can create large software system, in fact orders of magnitude smaller than it would be if that same application was create using a procedural programming language. It achieves this by starting with the concept of what it is

trying to create, in this case a database backed web application, and just requires the minimal "configuration" to make that work. Then additional data can be added and that deviates the application away from it's initial form in well defined and useful ways. So you can end up with a rich and powerful application with a very small amount of code.

Let me put this example into context. I long time ago I used to work on developing games on 8bit home computers. In that day hardware was very simple, particularly audio hardware. It was common to use what was called an Envelope to define sound effects and notes for music. The envelope consisted of a list of dozen or so integers expressed as bytes that defined various characteristics of the sound to be played such as how fast it's amplitude rose from silence called attack and the pattern of how it's pitch varied to create tremolo effects of musical arpeggios. Those dozen bytes defined all the sounds that you could create. You could actually get a surprising richness of sounds! If those bytes were put in an raw WAV file you would get much much less. But it is clear that you are never going to get the realistic sound of a piano never mind the expressiveness of the human voice from such a technology. Certain sounds, in fact most sounds, were just outside it's capability.

Looking again at the example of Ruby on Rails, we can see that although you can create some very rich and useful applications there are many things it cannot be used for. For example it couldn't be used for creating an Internet DNS server and it couldn't be used for creating a 3D first person shooting game. There are lots of things it couldn't be used for. That is not to be negative about it. It makes a very deliberate trade-off - it enables a small representation for systems which fit within it's problem domain.

There seems to be a trend in the IT industry towards solution/frameworks like these. They appear on face value to be a good solution to creating the necessary rich and complex applications that are required whilst managing the development costs and the specialist knowledge required. What can be wrong with this? With enough frameworks we can completely cover the space and then it's just a matter of picking the right one - right? However I believe they are not ideal. To understand why lets look at these three questions:

What to do when you hit a limit to their degrees of freedom?

When you hit a constraint in a given technology then if you are lucky it's a soft limit and you can drop out of the abstraction it offers to a lower level and solve your problem. Depending upon what you want to do that might completely sidestep any benefits derived from using it in the first place. It is essential that you have this capability though.

Can you understand enough about their scope up front to make a good choice for your future requirements?

Committing to technology is hard. You know that it's making you job easy up front or you'd already have passed it by -right? But what happens when you've got deep into you so-

lution and you hit a limitation? What happens if you hit a performance bottleneck in production and it is an inherent characteristic of the approach? The best you can do is to do your homework before committing and hope that some subtle issue doesn't bite hard.

What happens when different parts of system needs different technologies to be integrated?

Merging one or more technologies into a cohesive application can be hard. Different technologies come from different development teams and are based on different paradigms and approaches. The glue code to connect them is often nasty, hard to maintain and is hard to separate from your applications custom code.

So just like the ZIP compression is to MP3 let us look at software representation technology that isn't so narrow in scope. Let us look at one which aims to be able to represent pretty much any enterprise software whilst still concentrating the representation; reducing the amount of code. Unless you thought I'd had a last minute change of mind then you know what's coming next. Yep NetKernel. A common theme across pretty much everyone who has put a NetKernel system into production is the surprise at how little code there is. In fact one of our customers was a little embarrassed to share too much of the details of their solution to the client for worry that they'd be expecting much more for their money.

So how is this possible? That's a good question and one that isn't that easy to answer. We started out trying to change the economics of software in the narrow domain of XML message exchange systems back in last century and because of our backgrounds and the isolation from commercial pressures in the labs of HP through ruthless pursuit of the ideal we serendipitously discovered a new way. Over time we managed to distil this way into what we now call Resource Oriented Computing (ROC). An abstraction which when combined with a number of core technologies such as the standard module definition create basis for richly representing enterprise software very concisely, liberating the essence of software architecture from the nitty gritty details of code. ♠

Caching: How and why it works

Tony Butterfield

NetKernel's embodiment of Resource-Oriented (ROC) has the unique property of being able to transparently, efficiently and safely (i.e. not return stale values) cache computation. This is achieved through a process similar to [Memoisation](#) in functional programming languages but is generalized to be applied beyond functions without side-effects and even to functions that can change at non-deterministic times.

Wikipedia describes [caching](#) as:

“...a collection of data duplicating original values stored elsewhere or computed earlier, where the original data is expensive to fetch (owing to longer access time) or to compute, compared to the cost of reading the cache... Once the data is stored in the cache, it can be used in the future by accessing the cached copy rather than re-fetching or recomputing the original data.”

NetKernel has two distinct caches. Firstly a Representation Cache keeps response representations keyed against the requests that were issued for them. This is only useful for the usually idempotent request verbs of SOURCE, EXISTS and META. Don't worry right now about stale cache entries or non idempotent requests, NetKernel has a rich and powerful mechanism for expiry of representations that we will talk about in a minute. Before any request is processed by an endpoint, NetKernel will first check the Representation Cache to see if it can just pull on a pre-computed representation.

Secondly a Resolution Cache keeps request resolutions keyed against requests that are issued. Remember from our previous discussion of [Dynamic Resolution](#) how the resolution process can potentially become quite involved when large [modular address-space](#) is instantiated within a complex system. This cache acts to shortcut the resolution traversal. The Resolution Cache acts as backup behind the representation cache so that even when the response from a request cannot be cached NetKernel can start executing the request processing endpoint immediately.

Representation Validity

HTTP uses smattering of headers to define response expiration heuristics [Expires, Cache-

Control, Last-Modified, If-Modified-Since, ETags]. Part of the proliferation is legacy caused and partly because of the number of approaches to best optimising sharing best knowledge of potential expiry for different server implementations over the latency and cost of the network.

Luckily things are much simpler in ROC. A response has a single IExpiry interface:

```
boolean isExpired()
```

The expiry function is should be a monotonic function which should never return false after it has returned true. I.e. once expired, always expired.

Standard expiry implementations are provided:

- **ALWAYS_EXPIRED** - response should never be reused. The request is not idempotent or lacks referential transparency.
- **NEVER_EXPIRED** - response will always be valid
- **TIMED_EXPIRY** - response is valid until a given absolute time in the future.
- **DEPENDENT_EXPIRY** - response is valid until any sub-requests response is invalid. This is usually the default expiration function and ensures that expiration propagates to all dependent resources.

In addition custom expiry functions can be programmatically attached to a response allow rich and dynamic expiration mechanism such as watching for modified filesystem files or the [golden thread pattern](#) used to layer expiration semantics over external data sources such as relational databases.

Scope

Earlier in this discussion we said that the Representation and Resolution caches are keyed on the request. The actually fields used from the request are the resource identifier, the request scope, and the request verb. However, to improve hit rates we need to be a bit more subtle. NetKernel 3 employed a user settable flag "isContextSensitive" on each response, if false the response did not depend upon the request scope, i.e. the context of the request would not effects it's response. If true, the response depends upon the request scope and a request issued with a slightly different scope would be considered and computed independently. NetKernel 4 automatically determines how much of the requests scope is needed to bound the response. Not only is transparent but it is optimal.

If you found that last paragraph a little dry your certainly not going to be the only one. A possibly easier way understand the problem we solve here is to consider a real world request. If I should shout out "Hey Peter I want a cup of tea!" what will happen? If I'm sat in the office and Peter is in a good mood then this request will resolve to Peter Rodgers, CEO of 1060 Research (for want of a better identity) will jump up and make me a cup of my favorite tea. In this example Peter (tea making endpoint) was resolved in the scope of the office. He used his current mood (scope) used my tea preferences (scope) to determine his response. (Of course real world cups of tea are not cachable but lets pretend they are.)

Now if I change my scope by going out into the street and shout "Hey Peter I want a cup of tea!" what will happen? If I'm lucky enough to get a resolution of this request from some passing Peter then they are not going to have my tea making preferences in scope and this request will need to be re-evaluated. Let's say I ring in to the office to speak with Peter and ask for a cup of tea, now my scope is different however after a puzzled pause Peter says "The cup I made you earlier is still on your desk". Pulled back from the cache I get my cup of tea without any effort. Peter didn't care about my location (scope) to know what tea I wanted. Ok I admit it, that was slightly contrived!

Temporal locality

One of the things we came to realize was how effective caching was. We thought that an inevitable consequence of high level of abstraction that ROC brings would be a performance overhead. To quote the [SOA Manifesto](#) "Flexibility over optimization", we expected a compromise to gain the malleability and scalability of ROC. But in actuality we got increased performance too. Even small caches can have big effects on performance. It turns out that this is due to [Temporal locality](#). The real world has tendency for requests to form normal distributions with many resources being used again and again over a period of time and only a few that are one offs. Even with unique requests on the edge of a system the internals often gain a lot from caching. We find real world systems typically reduce computation by 50%.

I hope this posting has helped explain some of the magic behind caching in NetKernel. ♠

Visualizer – Time Machine Debugging

Tony Butterfield

[Video: Visualizer](#)

Over the summer I've been working upgrading NetKernel's Visualizer ready for a major new release planned for the autumn/fall. All this work combined with comments from various folks set me thinking that not much has been said about this critical tool which must appear so alien on first approach.

The Visualizer technology was developed along side the NetKernel 4 kernel in the summer of 2007. After earlier experiments with more traditional breakpoint based approaches to debugging we realized that the ROC abstraction was at just the right level to allow a much more powerful debugger to be implemented. In a nutshell the difference between the visualizer and a regular debugger is that **the visualizer captures everything that happens over a period of time** rather than simply showing the instantaneous state at the time the breakpoint is activated.

Capturing everything probably sounds crazy/impossible right? But when you consider the pervasive [immutability](#) of state in [Resource Oriented Computing \(ROC\)](#) it starts to sound feasible and that combined with the naturally more granular approach that ROC encourages over object oriented and functional approaches I hope you see that it is indeed possible! (If your still not convinced then download NetKernel now and give it a try - it'll take you less than 5 minutes to get it up and running.) What's more, capturing visualizer traces have close to **zero performance overhead** to an executing system. Again this is due to the immutability of state - capturing simply involves hanging on to state for longer than it otherwise would be - so, yes, this does take extra heap space.

The real win with the visualizer is that **the trepidation of overstepping the unknown critical line of code** is gone. No more spending lots of time elaborately setting up some situation and carefully stepping through it in the debugger only to find that you were a bit zealous with that step-over button or you realize you realize that what you've just spend and hour tracking down was a symptom and not the cause. So maybe the [physicists can't pull off time travel](#), but you can with the help of the visualizer. Time machine debugging really means stepping back in time as well as forward. I really find it hard going back to a regular debugger.

Other benefits of the visualizer include:

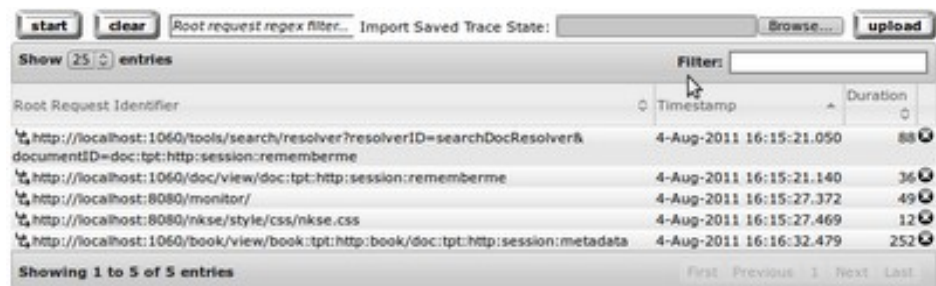
- Realtime profiling - with NetKernel Enterprise Edition all timings are captured so local and elapsed times of all requests are captured and can be explored to look for hotspots.
- Testing - visualizer traces can be programmatically captured and introspected enabling rich testing of not just what but how and why.
- Modeling - capturing the full details of how requests are processed can lead to some interesting visualisations!
- Support - Visualizer traces can be saved and re-loaded aiding rapid remote support and all the necessary information for bug fixing.
- Evaluation happens in real-time - so there are no "[heisenberg effects](#)" and debugging happens after the fact at your leisure.

Visualizer Walkthrough

Let's have a look at some of the key aspects of the implementation. In a later post I'll maybe go into more detail if there is interest. I'm going to show the new version - the core concepts and layout remain unchanged. The main differences are in refinement of the UI and new extended views and tools.

First step is to enable the visualizer and capture some traces. The main page lets you do this as well as manage the traces you have captured:


Figure 1: Table of captured request traces



Root Request Identifier	Timestamp	Duration
http://localhost:1060/tools/search/resolver?resolverID=searchDocResolver&documentID=doc:tpt:http:session:rememberme	4-Aug-2011 16:15:21.050	88
http://localhost:1060/doc/view/doc:tpt:http:session:rememberme	4-Aug-2011 16:15:21.140	36
http://localhost:1060/monitor/	4-Aug-2011 16:15:27.372	49
http://localhost:8080/nkse/style/css/nkse.css	4-Aug-2011 16:15:27.469	12
http://localhost:1060/book/view/book:tpt:http:book/doc:tpt:http:session:metadata	4-Aug-2011 16:16:32.479	252

Clicking the start button will start the capturing of traces, clicking it again will stop. Each entry in the table is one captured trace which corresponds to the full execution state of one root request (an external event/request injected into NetKernel through a transport) from start to end. You have the option to capture all requests or specify a regular expression filter on the root request identifier, so for example only capture requests from a particular transport or to a particular application.

It is worth stating here, for those less familiar with ROC, that the root request usually initiates sub-requests and those sub-requests initiate further sub-requests down to arbitrary depths into your application. The resource oriented approach isn't just surface deep. It is this tree of sub-requests that form the captured trace.



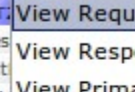
A screenshot of a context menu with three options: 'View Request Trace', 'Save to File...', and 'Delete'. A mouse cursor is pointing at the 'View Request Trace' option.

Figure 2: View of request trace showing recursive tree of sub-requests

Physical Endpoint / Collocation	Space	Verb	Request Identifier	Duration (ms)	CPU %	Exp.	Response Representation
InternetTransportEndpoint							
InternetGateway	Falcon / Backend Interface	SOURCE	http://api.adapt-1050/vm/network/hg-hg-wireless-network-burne	36	0.45	fail	
EthernetTransportEndpoint	HTTP Bridge Overlay	SOURCE	ns:///ac/www/loc/http-session-manager-burne	35	0.08	fail	
ApplicationGateway	Control Panel (private)	SOURCE	ns:///ac/www/loc/http-session-manager-burne	35	0.11	fail	
ControlPanelSource	Control Panel (shared space)	SOURCE	active-control-panel-on-gpu-partia-request-dbg-vhs/transport	-	0.39	fail	Response (impl)
VideoInputEndpoint	VideoControl (shared request)	SOURCE	pjs/geturl	0	0.08	fail	WebP format response (shared OnyxImpl)
CameraInputEndpoint	Control Panel (shared space)	META	on-geturl	0	0.01	fail	Meta Info
EthernetTransportEndpoint	Control Panel (shared space)	SOURCE	ns:///advice/webcam/gateway/openapp-fusion	3	0.04	fail	
InternetAccessGateway	NKSE / Trunk (private)	SOURCE	ns:///advice/webcam/gateway/openapp-fusion	3	0.03	fail	
HTTPRequestEndpoint2	HTTP Request Response Space	SOURCE	httpRequest///control/webcammanagementC-available	0	0.03	fail	Cookie
BaseImageEndpoint	NKSE / Trunk (private) / OverlaySpace	SOURCE	ns:///advice/webcam/gateway/openapp-fusion	3	0.03	fail	
HTTPSessionEndpoint	HTTP Session Space	SOURCE	session-id/burne	0	0.01	fail	
AuthenticationEndpoint	NKSE / Trunk (private)	SOURCE	active-pjs/geturl	1	0.08	fail	ContentRepresentation
BlobAccesser	mermaid data store (private)	SOURCE	pjs/pjs/geturl	1	0.08	fail	Available Memory Stream
IStream	NKSE / Trunk (private)	SOURCE	ns:///ac/pjs/geturl	0	0.03	fail	Available Memory Stream
IPublisher	Layers	SOURCE	ns:///ac/pjs/geturl	0	0.08	fail	Memory
NonInteractiveSource	System Services (private)	SOURCE	noninteractive///webcam/data-store-on-gpu-partia-request-dbg	0	0.03	fail	String
GetFromCache		SOURCE	active-pjs/id-test/csp	0	0.08	fail	Available Memory Stream

quest execution tree laid out not just the current call stack. A call stack view (from the perspective of a particular sub-request) is available as are various other views. Each row represents one sub-request, the endpoint that handles it and the response returned. The columns in the request trace view are:

1. The call tree with the endpoints that are invoked for each request.
2. The address space which hosts the endpoint
3. The request verb
4. The request identifier
5. The elapsed duration of processing the request
6. The local time spent inside the endpoint not including sub-requests
7. Expiration status of response
8. Representation of response



The screenshot shows the Visual Studio Code interface with a REST client request. The 'View Request Details' context menu is open, displaying the following options:

- View Request Details
- View Response
- View Primary
- View Callstack
- View Trace Rooted Here
- View Expiry Determinants
- View Scope Determinants
- Compare Caching

Request	
Requestor:	MapperOverlay
Mode:	Synchronous
Verb:	SOURCE
Identifier:	active:java+class@org.netkernel.nkse.search.endpoint.SearchResolverAccessor+ /documentID+mode@redirect+resolverID@httpRequest%3A/param/resolverID
Request Scope:	<input type="radio"/> Search Application (private) /overlay/space /mapper/space <input type="radio"/> Search Application (private) /overlay/space <input type="radio"/> Search Application (private) /overlay/space <input type="radio"/> Search Application (private) <input type="radio"/> Search Application <input type="radio"/> Control Panel Wrapped space <input type="radio"/> Control Panel Wrapped space <input type="radio"/> Control Panel (private) <input type="radio"/> Control Panel <input type="radio"/> HTTP Bridge Overlay <input type="radio"/> HTTP Request Response Space <input type="radio"/> Fulcrum / Backend /rootospace
Requested Rep:	java.lang.Object
Priority:	NORMAL
Headers:	[none]

Figure 3: Request Details

I don't want to go into any discussion here about each of the displayed fields. The point I want to make is that all the details are there captured.

Figure 4: Resolution Details

The resolution process is mechanism that the NetKernel kernel uses to locate an endpoint to evaluate a request. The detail here shows how the request scope is systematically interrogated until a resolution is found. If a match is found the endpoint is then used for evaluation. For more details on the scope concept see my earlier post on [Dynamic Resolution](#).

Resolution	
Resolution Trace:	<input type="checkbox"/> Starting search in [Search Application (private) /overlay/space /mapper/space] <input type="checkbox"/> Entering import [Search Application Private Library] No Match on [Fileset] <input type="checkbox"/> Entering import [XML / Core Library] No Match on [Import [urn:org.netkernel:xml:core]] <input type="checkbox"/> Entering import [Wiki Resource Model] No Match on [Import [urn:org.netkernel:wiki:core]] <input type="checkbox"/> Entering import [Wiki Resource Model] No Match on [Import [urn:org.netkernel:wiki:macros]] <input type="checkbox"/> Entering import [System Services] No Match on [Import [urn:org.netkernel:ext:system]] <input type="checkbox"/> Entering import [Lang / Freemarker] No Match on [Import [urn:org.netkernel:lang:freemarker]] <input type="checkbox"/> Entering import [Lang / Groovy] No Match on [Import [urn:org.netkernel:lang:groovy]] <input type="checkbox"/> Entering import [Layer 1] No Match on [WormholeAccessor] No Match on [DataSchemeAccessor] No Match on [FileSchemeAccessor] No Match on [ForEachAccessor] No Match on [SpaceHDSAggregator] No Match on [JarSchemeAccessor] No Match on [HDSXPathAccessor] No Match on [GoldenThreadAccessor] Match on [JavaRuntimeAccessor]
Physical Endpoint:	JavaRuntimeAccessor
Logical Endpoint:	layer1.JavaRuntime
Resolved Scope:	<input type="radio"/> Layer 1 <input type="radio"/> Search Application Private Library <input type="radio"/> Search Application (private) /overlay/space /mapper/space <input type="radio"/> Search Application (private) /overlay/space <input type="radio"/> Search Application (private) /overlay/space <input type="radio"/> Search Application (private) <input type="radio"/> Search Application <input type="radio"/> Control Panel Wrapped space <input type="radio"/> Control Panel Wrapped space <input type="radio"/> Control Panel (private) <input type="radio"/> Control Panel <input type="radio"/> HTTP Bridge Overlay <input type="radio"/> HTTP Request Response Space <input type="radio"/> Fulcrum / Backend /rootospace

Physical endpoint / callback	Verb	Request identifier	Duration [ms]	Response representation
org.netkernel.layer1.endpoint.JavaRuntimeAccessor	SOURCE	active space class org.netkernel.nsa.search.endpoint.SearchResolverAccessor+data	14	null
HTTPRequestEndpoint	SOURCE	httprequest / parent/resolved	26	string
HTTPRequestEndpoint	SOURCE	httprequest / parent/resolved	0	string
spaceAggregator	SOURCE	active spaceAggregator+url http://www.baylib.com/search/seekingba.html	17	void
URLConnection	TRANSFER	active url http://www.baylib.com/search/seekingba.html	0	InputStream
URLConnection	SOURCE	active url http://www.baylib.com/search/seekingba.html	0	string
HTTPResponseEndpoint	SINK	httpresponse / redirect	0	null

Figure 5 Endpoint Evaluation Details

The endpoint details section is only shown if the endpoint is evaluated. Often times the response can be pulled from cache. Details of any direct sub-requests issued are shown along with their expiration and duration, both useful for determining correct and efficient operation.

Response
Representation: null
view
Headers: [none]
Expiry: AlwaysExpired
IsExpired?: true
Cost: 50364 of which 2327 is local
Response (yellow area indicates scope used in processing request)
Scope:
Search Application (private) /overlay/space /mapper/space
Search Application (private) /overlay/space
Search Application (private) /overlay/space
Search Application (private)
Search Application
Control Panel Wrapped space
Control Panel Wrapped space
Control Panel (private)
Control Panel
HTTP Bridge Overlay
HTTP Request Response Space
Fulcrum / Backend /rootospace

Figure 6 Response Details

Again all the details of response are here. **Nothing is hidden.**

I hope this post gives a good overview of the visualizer. I plan to talk in more detail at a later stage about some of the new views that we'll be releasing soon and what has motivated them. In writing this post I notice there are a lot of details of the NetKernel implementation of ROC that are not really talked about elsewhere, for example, what does cost on the response mean? what does the expiry field really show? Let me know if there are other dark corners you would like me to shed light on. ♠

Ending the Absurdity

Tony Butterfield

"If at first, the idea is not absurd, then there is no hope for it."

– ***Albert Einstein***

Peter and I were talking the other day about how absurd the ROC (resource oriented computing) model can appear. This came about because I'd become re-acquainted with the inner workings of the NetKernel kernel whilst implementing a number of low-level changes for the upcoming version 5 release.

How can something so convoluted that every atom of computation resolves to an end-point through a set of arbitrary and dynamically interrelated address-spaces by an abstract identifier possibly work? These computations are then cached using a deep dependency based validity model and their computation costs determined and accumulated to minimize future computation. Surely all this complexity can't work, or at least can't work well?

Now, if you're thinking what I thought on first seeing Einstein's quote then you're going to be saying to yourself "yeah sure, but I bet that there is no hope for most of the absurd ideas either!" And yes I can't argue with that, but it's the converse that's interesting; why is it that most enterprise software lacks any absurdity? Why is it in general that new hot trends and technologies are so conservative in ambition and only willing to make single safe steps forward? Why do the same old ideas get rehashed in new clothing again and again destined to fall short?⁸

⁸ I understand some of what I've said with regard to the general lack of ambition in software may be controversial. (Please comment.) I am of course generalizing; there is lots of great software out there often hidden away quietly doing it's job well. But it does frustrate me that we as a society could be benefiting hugely if we could more economically create and maintain large software systems. We need to remove the inherent brittleness that goes right to the foundations. That is [what ROC does](#).

I find the conservative nature of the software industry very difficult to understand. We are working in an industry with the most fluid of materials: concepts and information. And of course, with NetKernel, we have taken a very different tack. So trying to think of some reasons, I came up with the following:

Big business gets stuck with the innovators dilemma - big new ideas that disrupt existing markets are not usually fostered.

Good ideas are in short supply - if there is one takeaway from entropy it's that there are many many more disorganized, broken or useless permutations of state than good and useful ones.

Abstract is abstract - there is no getting away from the fact that NetKernel's power and ultimate flexibility comes at a price. That price is the need for a conceptual leap. The dictionary definition doesn't just describe abstract as "expressing a quality or characteristic apart from any specific object or instance"⁹ but also "difficult to understand; abstruse". Yet abstraction is absolutely needed for us to make sense of the world. From the day we are born we are finding and refining our abstractions.

Right place, right time...

Both Peter and I had been working independently in different organizations (I in a large utility company, and Peter in Hewlett Packard) on developing approaches to reducing the cost of software. XML was showing itself to be a powerful generic data language and was gaining traction to the extent that a rich set of tools around it were developing. When we came together at HP our joint backgrounds in physics, innate curiosity, irreverence to convention and desire for unification led us down a new path. We saw many distinct strands that could be tied together. We had/have a creative tension. Peter always has an eye on the big picture, the ability to pick apart an obscure technology or standard and see what makes it tick, and most importantly the drive to ensure what needs to be done gets done. I, on the other hand, have a desire to create deep symmetry, an obsession for optimization (from my misspent youth developing 8bit games) and an eye for aesthetics and design. Together we created NetKernel.

With the imminent release of NetKernel 5 we have a solid, rounded platform, battle hardened and embodying the state-of-the-art ROC. There is still work to do however. We must now turn our attention to the learning curve and make the abstract clear, uncomplicated and obvious. If you are using NetKernel now you are one of the self selecting few. For every one

⁹ dictionary.reference.com

of you here there are many more who have bounced off the absurdity or the subsequent learning curve. This is our next challenge.. ♠

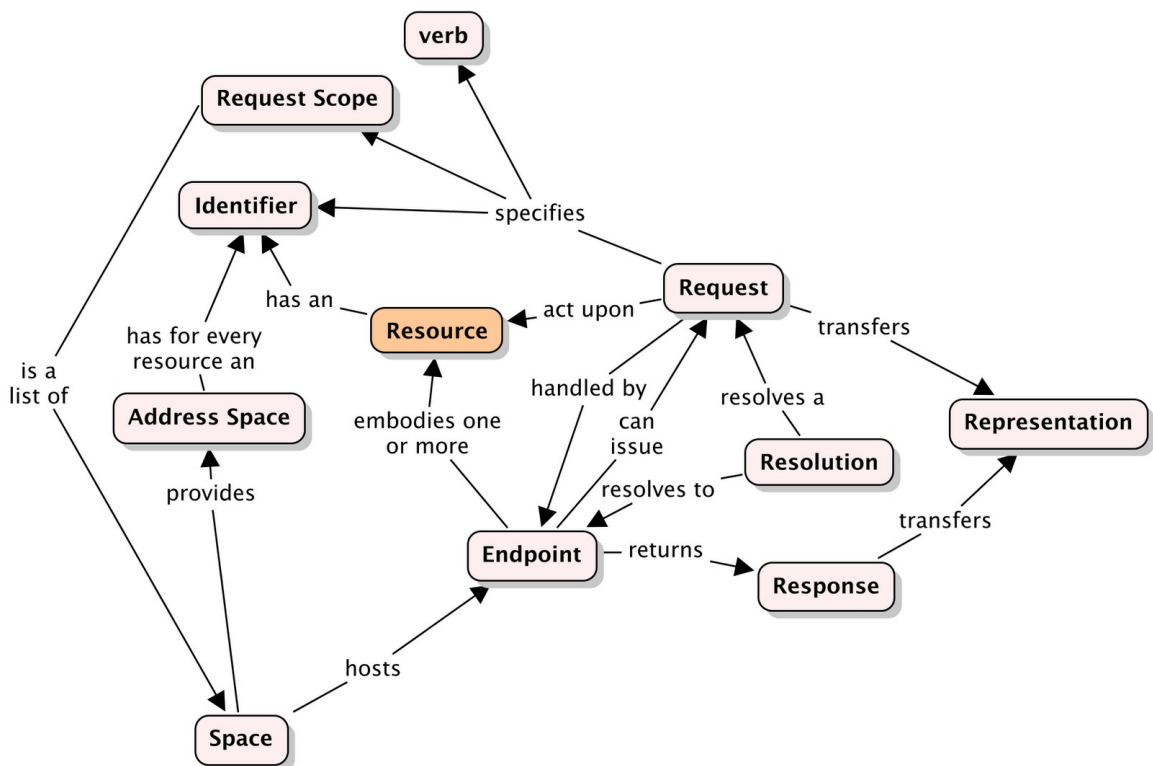
ROC Concept Maps

Tony Butterfield

Listening to the people I've spoken to this week then a picture may not be worth a thousand words. But those folks tend to prefer the command line over the GUI. If you're in that camp you know where the source code to NetKernel is and you've probably read it. This post is for the rest.

Following on from [Essence of ROC](#) I've decided to do a series of [concept maps](#) of the ROC principles. Concept maps are pretty good at visually showing relationships between concepts. In fact they seem pretty close to [RDF](#) when represented as a set of proposition triples. Each concept is shown as a rounded box with the concepts name centred within it. Connecting between the concepts are linking phrases. These can be anything that makes sense within the domain. Unlike [UML](#) and more structured modelling languages there are no rules.

So first let's look at a map that is based on the concepts we explored earlier in Essence of ROC. We ask the question "What is a resource?" and this forms the centre of our map with resource highlighted in orange.



We can see that this visual representation is pretty compact and although it potentially fails to capture some of the fine detail it gives a clearer high-level view. Of course we could add extra fine concepts to the map but that might hinder more than it helps. So my tendency will be to present a clear concept map and a minimal associated prose to fill in a few blanks if necessary. It might be nice to have little comment boxes like UML and [nCoDE](#) but unfortunately the [tool I've been using](#) doesn't support them.

Compare this map to the mentioned principles with their associated proposition triples. (The observant of you will note that I've dropped the last principle about transreptors; thank you for those who pointed out it that it might be better left off, with hindsight I realise you were right because the concept of transreptors much better fits into the subsequent section.)

Everything is a resource

- Resource has an Identifier

Endpoints embody resources in software

- Endpoint embodies one or more Resource

Requests act upon resources

- Request act upon Resource
- Request handled by Endpoint
- Endpoint returns Response
- Request specifies Identifier
- Request specifies verb

Representations transfer state

- Request transfers Representation
- Response transfers Representation

Spaces host endpoints

- Space hosts Endpoint
- Space provides Address Space
- Address Space has for every resource an Identifier

Request Scope determines the resolution of a request

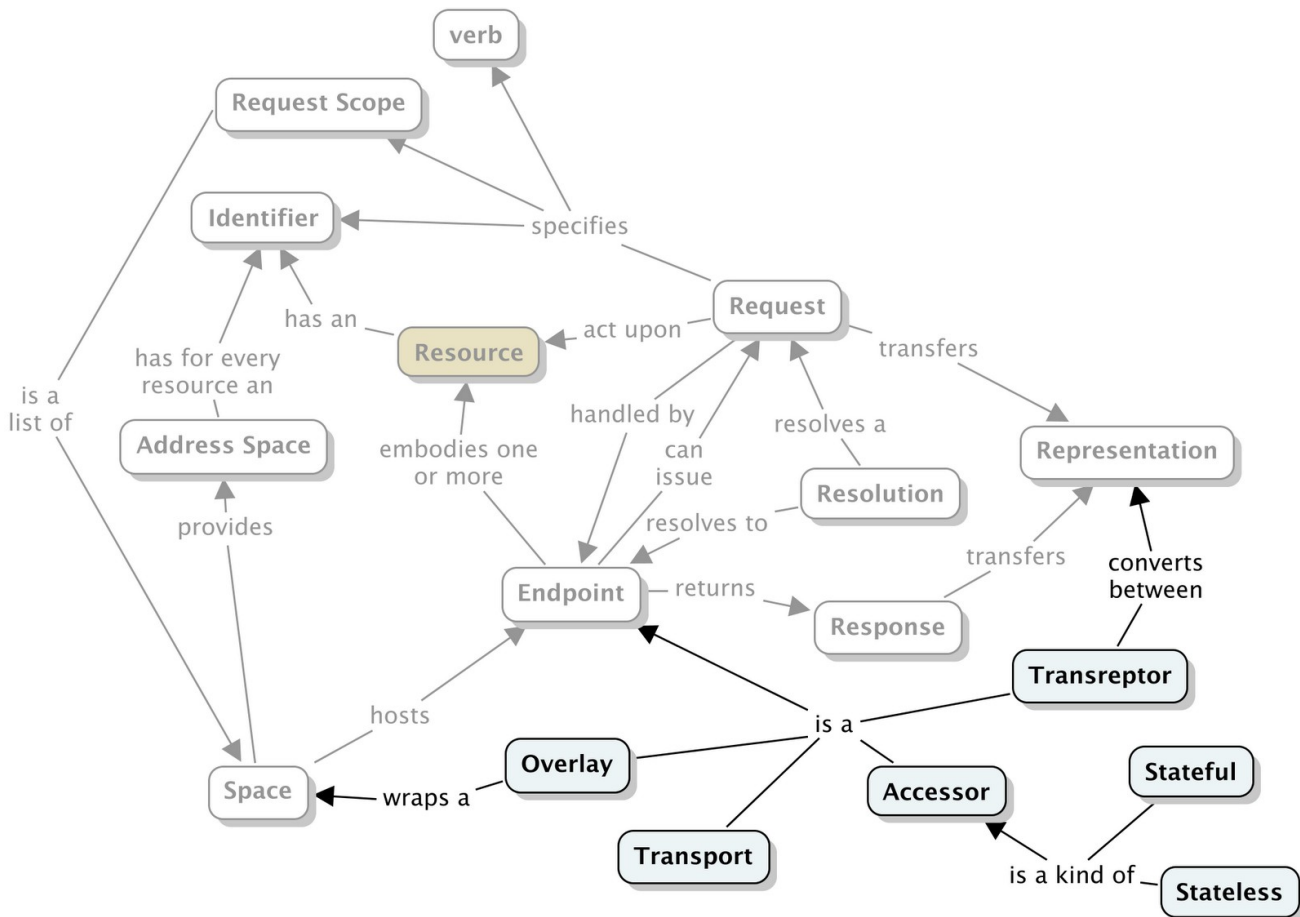
- Resolution resolves a Request
- Request specifies Request Scope
- Request Scope is a list of Space
- Resolution resolves to Endpoint

Endpoints can issue requests

- Endpoint can issue Request

Exploring the types of endpoint

Now we have the foundation as a concept map, I want to push forward and add in some more of the concepts. First up let us explore some of the variants of endpoint:



Endpoints come in four varieties, Overlays, Transports, Accessors and Transreptors (yes this is where it fits).

[Overlays](#) wrap a space, as such they manage access to the wrapped space from the host space. Overlays have many uses including implementing a simple import of one space into another.

[Transports](#) are endpoints which don't embody any resources and no requests can resolve to them. What they do is receive events from outside the address space and convert them into requests and issue them into their hosting space.

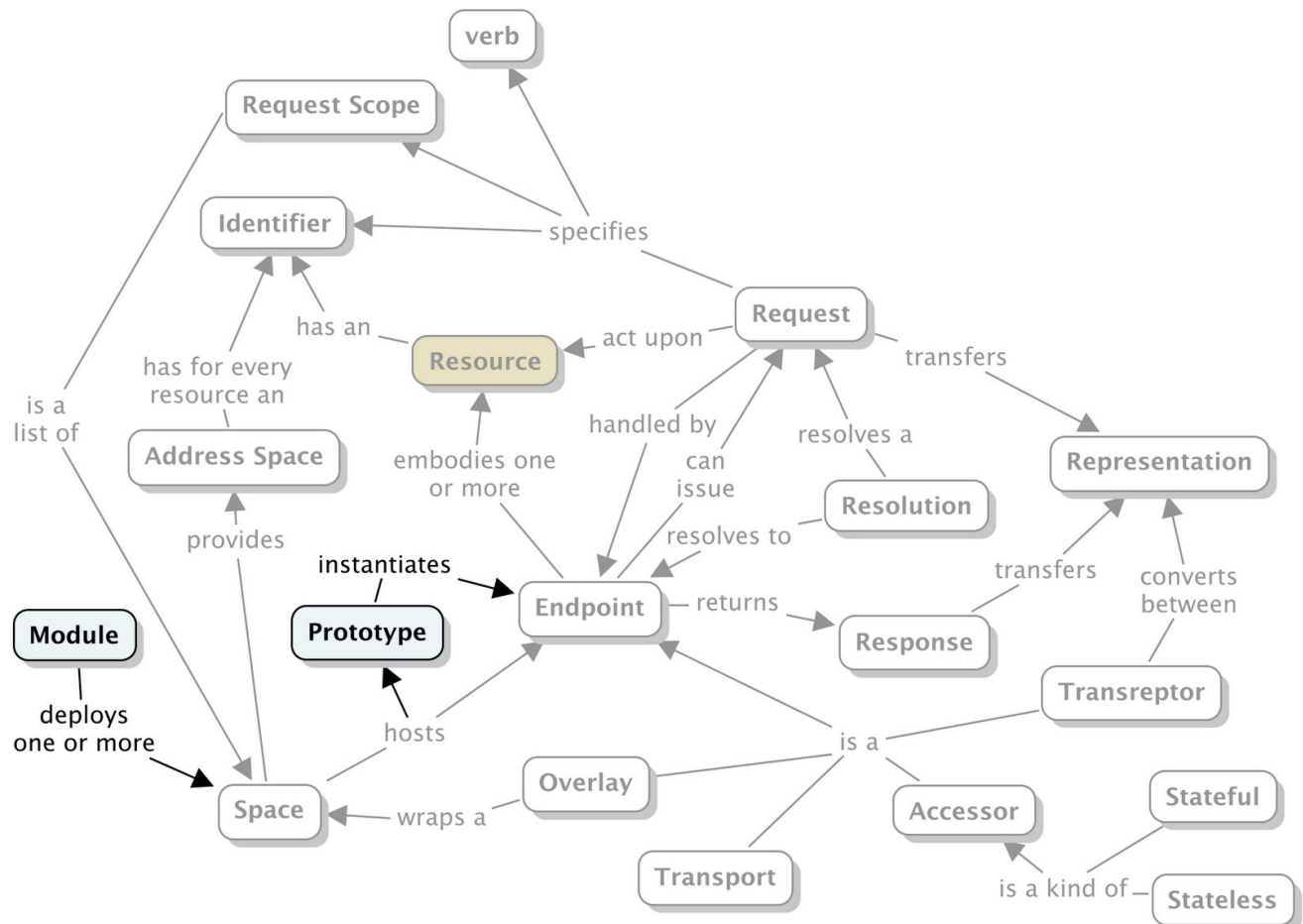
[Transreptors](#) are again endpoints that don't embody any resources. They convert representations from one form to another. Their purpose is to adapt representations from what an im-

plementing endpoint provides to what a requesting endpoint wants. A simple example being a compiler that takes a sequence of characters and transrepts to an abstract syntax tree.

Accessors, in a sense, are the most obvious of the endpoints. They do directly provide an embodiment of one or more resources. For example a File accessor maps the files within a filesystem to resources within the host space with identifiers starting with "file:" The file accessor resolves SOURCE, SINK, EXISTS and DELETE requests which have a direct effect on the file on the filesystem. The file accessor is an example of a stateful accessor because it models a resource which has state. Stateless accessors typically model transformations or language runtimes. These embody derivative resources which depend upon the state of other resources rather than any state which they directly contain. For example consider the Groovy runtime accessor which has no state but which provides the response from executing a given groovy program. Usually these stateless or service accessors only implement the SOURCE verb.

Minor practicalities of Modularity

This section has a couple of concepts (modules and prototypes) relating to the the modularity of systems built around the resource oriented concepts.

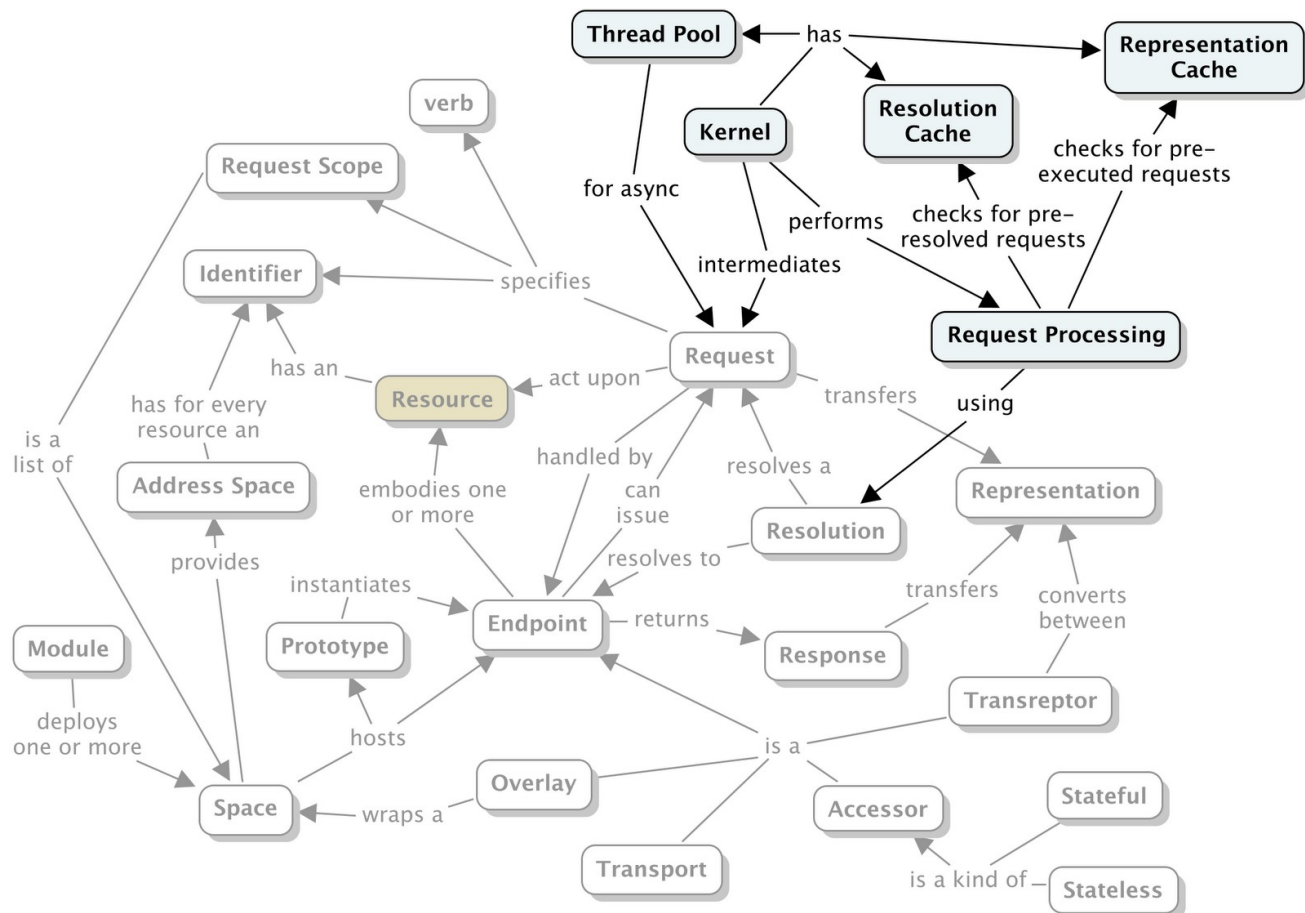


A module is the unit of deployment. It deploys one or more spaces with their endpoints. NetKernel supports dynamically deployable modules and also pluggable module implementations. Currently only one module implementation technology is supported, this is the [Standard Module](#), though other non-supported implementations such as the legacy NetKernel3 Classic Module and a low-level Java Module are available.

Spaces can host [prototypes](#) in addition to endpoints. A prototype exists to allow the creation of new endpoints based on an implementation provided in a library space. It is often necessary to create new endpoint instances rather than simply issue a request to an existing endpoint when the endpoint depends upon it's location, i.e. an overlay or transport.

So What Makes it Tick?

This section shows how the kernel at the heart of NetKernel lies a kernel which acts as the intermediary middleware between requests.



The kernel mediates between requests. Although we logically think of endpoints issuing requests to other endpoints the reality is that endpoints have no direct coupling between each other. What actually happens is that requests are issued to the kernel and it performs

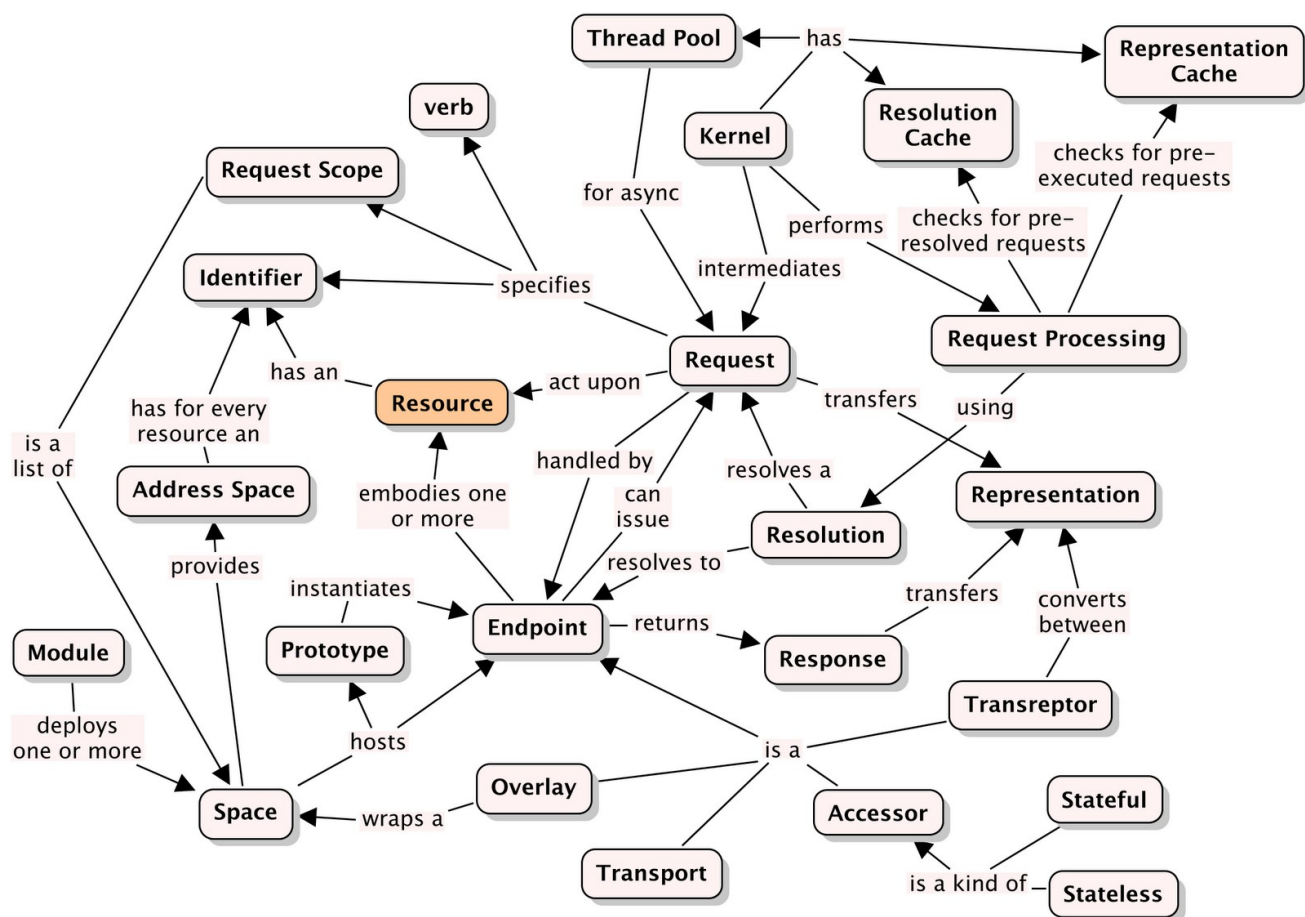
resolution of the request to determine an endpoint to process the request based upon the request scope which is contained with the request. When the request has been resolved it is then forwarded on to be evaluated by an endpoint. If an endpoint doesn't return a response representation that the requesting endpoint desired then the kernel will also attempt resolution and evaluation of a transreptor to change the representation.

The kernel also enables the implementation of asynchronous requests by handing of a request into a queue that is then serviced by a worker thread pool. This enables the calling endpoint to perform many request concurrently.

One of the big wins with ROC is the ability to [cache results of computation](#). The kernel maintains a representation cache which is checked before it considers forwarding a request to an endpoint. In addition the kernel maintains a resolution cache which is checked before it attempts to resolve a request.

Pulling it All Together

With nothing more to be said, here is the complete concept map covering everything we've discussed:



The Future is Coming on

Tony Butterfield

Recently I've been feeling dissatisfied with the IT industry. From the outside people think it's all moving so fast and is full of relentless innovation towards the future. From the inside it feels at least in part like a stagnating pool with innovation increasingly confined to narrow niches. In this article I want to explore my thoughts further.

The first real practical computers were realised in the 1940's though the theoretical foundations were established a decade earlier. At that point the separation of hardware from software through universal computers was established. Hardware has been on a steady march forward since that time driven by discoveries in science and advances in materials. This is epitomised by the legendary Moores law though it's not just processing power that's increasing, storage capacity and network speeds are too, whilst power consumption and size are falling. Hardware is exciting! The internet of things is becoming real. Exciting gadgets and novel peripherals appear almost weekly and sometimes they even offer real benefits.

But all this contrasts with software. It's true that computer games are pretty amazing and completely unrecognisable in their ambition from what existed a decade ago. And smartphone apps show just how much creativity there is out there to leverage a personal computing platform. To top it all we have the world wide web the most amazing and rich system that humans have ever created. So why all the grumpiness? Well, let me tell you!

Serious overruns and large over-budget projects are commonplace and sometimes they fail completely. There is a lot of publicity after this happens in the public sector but in my experience this happens often in the private sector too. There is something about the scale of projects in terms of their scope and complexity that is non-linear. Also software often doesn't manage to take advantage of the advances in hardware. In simple cases such as rendering video streams and compression of data things work well but take a look at operating systems: how often does an hourglass pop up and a desktop freeze for (from the outside) inexplicable reasons? This is just on a standalone system, business systems are usually much worse with bad performance, poor usability and inability to evolve in a timely manner to the needs of their users. Also there is the issue of quality; there is an inevitable tension between quality and cost and as such there is never a perfect system because there is not infinite budget. However a man on the moon is testament to the fact high quality can be bought. Quality like many of the other attributes of software can be a function of cost and it these economics that drive software. The high price and unpredictability is the root of the problem. As a side,

economics is not the whole story, the classic Brooks book *The Mythical Man Month* shows how some things just take time.

There are many factors that lead to the high cost of software. Some of them are intrinsic and unavoidable such as the complexity and dynamics of requirements and necessary overheads of teams as described by Brooks. Others, I will argue, are caused by the way we work - software engineering. It's interesting to compare how software engineering is described in Wikipedia with straight engineering:

Software engineering is the application of a systematic, disciplined, quantifiable approach to the design, development, operation, and maintenance of software, and the study of these approaches;

vs,

Engineering is the application of scientific, economic, social, and practical knowledge, in order to design, build, and maintain structures, machines, devices, systems, materials and processes.

Note the focus on approach and on the non-absolute metrics of systematic, disciplined and quantifiable in software engineering. Whereas engineering in general uses scientific, economic, social and practical knowledge to achieve something. I can certainly testify to the general obsession with the construct vs the artifact as described by Rich Hickey in this great presentation. (The construct being the code/process and the artifact being the developed system.) I remember being shocked as a graduate going through ISO9001 checks within the large organisation I worked for and getting told "it doesn't matter what we do, it just matters that we do what we say and can say what we do."

One of the effects of this approach is the reduced creativity that can be applied to the artifact. Energy goes into things like creating an elegant code construct or searching for a coding language with more succinct and expressive syntax but usability and stability under extreme operating condition is neglected. In addition the development model is still very artisanal. Highly skilled developers are pooled and work adhoc on various facets of functionality. A framework or key technologies (tools) may be chosen at the start but the bulk of the construct is created from the ground up for each project in a bespoke way. It's hard work building skyscrapers if you are placing atoms. As long as I have been working in the industry (23 years) reuse has been touted as necessary and important but never achieved at an enterprise level. Component models have never worked well enough and object reuse quickly fails due to the difficulty of coupling. This has lead some to give up on reuse altogether. Of course we've always had reuse of point technologies and off-the-shelf frameworks can help too but nothing that runs deeper into "business logic" code.

My view is that we are approaching the equivalent of an industrial revolution in software. The economics must change such that we can build way more complex systems reliably that meet the needs of their users and can be evolved efficiently. We need specialist roles with certification. We need companies creating high quality engineered components that can be click-fitted together and selling them in a sustainable market. Systems will be engineered with hard metrics and quality assurances built in. Architecture will be separated from code such that constraints can be applied orthogonally. This revolution will be technology driven - by a technology that truly delivers a component architecture that can span and unify the range of technologies we use today and scale from the nano-scale to the global. That technology is Resource Oriented Computing and its first instantiation is NetKernel.

Of course that's a bold claim! I don't think NetKernel is the final solution but it is a seed. What it offers is a component model that is scale invariant - i.e. you can sequence integer additions, through transforming XML all the way to load-balancing network requests in a data center within the same abstraction. The stateless REST based model using immutable representations leads to excellent linear scaling. Compositional components built within the abstraction itself can allow sequencing and orchestration of other components using techniques of imperative, functional, state machine, data-flow, or whatever you can think of. Address spaces and URI indirection and common REST based constrained interfaces allow components to cooperate even when they don't know about each other and were never designed to work together. And then there is caching - every computation that has been performed is uniquely identified and can be reused if applicable leading to what we call pseudo-static performance in dynamically reconfigurable systems.

This is quite different from mainstream approaches to software today and, if that is where you are then, there is some hill climbing to be done to see the benefit. However give NetKernel to someone fresh out of college and they'll run with it. When used as it was designed to be used NetKernel delivers metrics that are, frankly, unpublishable - at least at the moment. However there are a growing group of entrepreneurs who are betting their socks on it. There is a lot happening this year and I'm really excited to be a part of it. ♠

ROC: Step away from the Turing tape

Peter Rodgers, PhD

You might have caught wind of this thoughtful blog post by "Uncle Bob" Martin¹⁰ earlier in the week... <http://blog.objectmentor.com/articles/2010/07/05/software-calculus-the-missing-abstraction>¹¹

In which he points out that the conventional software world might not be the whole picture - his metaphor is that software today is like mathematics was, when it only conceived of algebra and yet whole new vistas opened with the dawn of calculus. His sentiment appears to have resonated with the wider community.

He ain't wrong. Nor is he the first to feel this. In the early 2000's philosopher Toby Ord, a graduate student of Jack Copeland, offered a similar view in his review article: "Hyper-computation computing more than the Turing machine"...

<http://arxiv.org/pdf/math/0209332>

...where, in the introduction, he says:

"In many ways, the present theory of computation is in a similar position to that of geometry in the 18th and 19th Centuries. In 1817 one of the most eminent mathematicians, [Carl Friedrich Gauss](#), became convinced that the fifth of Euclid's axioms was independent of the other four and not self-evident, so he began to study the consequences of dropping it. However, even Gauss was too fearful of the reactions of the rest of the mathematical community to publish this result. Eight years later, [János Bolyai](#) published his independent work on the study of geometry without Euclid's fifth axiom and generated a storm of controversy to last many years."

So, why am I pointing these references out to you? Well, apart from being cajoled to "pipe up" by my friend Mr B. Sletten, because, I've been thinking about this stuff for a long

¹⁰ Uncle Bob's website: <http://www.cleancoders.com>

¹¹ "This software transformation, whatever it is, is coming. It must come; because we simply cannot keep piling complexity upon complexity. We need some new organizing principle that revamps the very foundations of the way we think about software and opens up vistas that will take us into the 22nd century."

time. To get to where we are with NK and ROC, we've been embracing some heretical beliefs for over ten years.

There is no room to provide the detailed background, and besides we all have real work to do, but for the sake of at least tantalizing you, here are some of the cornerstones...

Information is Platonic - all possible information resources are "out there". Software development is the art of introducing and constraining context in order to reify (make real) tangible instances of the Platonic resource set. Identity is relative to context and may be traded - identity shrinks as context is constrained. There is an innate duality between identity and representation.

To the software community, perhaps the biggest heresy is:

There are an infinite set of possible Turing engines with a corresponding set of valid code (computable identifiers); known respectively as "languages" and "programmes". Ultimately, nobody who uses the solution cares. Put another way. There are no prizes in the "My language is better than yours" competition - in fact there are no judges (only priests).

So, in Uncle Bob's analogy, language is the "Algebra" of the classical software paradigm used in the mainstream today, in that we solve information problems by working within a given Turing dialect. One kind of Turing engine, one representational form for code and ultimately a world that maps into itself. You might call this a "Turing monoculture".

The [Turing-Church thesis](#) states that there's nothing inherently missing from a single Turing framework. Any given Turing machine can compute exactly the same as any other. But "living on tape" leaves no room for externalities and emergent state to react with and influence the context.

As Ord points out, misconceptions have grown up about computability and Turing completeness. Misconceptions that Turing never held; he was well aware that a Turing machine can only compute what a mechanistic mathematician can, with no scope for intuitive leaps. By which, I would interpret "intuitive leap" as meaning: there is no scope for accumulated prior knowledge to be applied to shortcut the route to future required state.

In his review, Ord does a good job of summarizing a range of theoretical Hypercomputers. That is, computational models that *are* able to compute more than the Turing machine. [I'll let you Google "hypercomputing" for examples of real world systems that in everyday life cannot be "Turing machine computed" and yet have been reified some how. A good example is how some simple curves have no computable first order derivative but you can draw representations of them.]

For example, what if you had what Ord calls a "Turing Machine with Initial Inscription". ie a Turing machine that contained within itself a lookup table for all possible input states that the machine could ever experience in production. Impossible, yes. But this machine would actually never have to "compute" anything. It would have the answer in advance for every external state change.



Turing Machine with Turing tape by Mike Davey¹²

So, where does ROC come in? ROC is a two-phase contextual abstraction for computation. The first phase is resolution - taking a resource identifier and using the context to either find a representation or, if not present, to find a Turing engine able to reify the resource. The second is the classical execution of code - but the twist is that during the execution you can, and invariably do, step back into the address space and issue further requests, which results in a cascade of the two-phase cycle.

Using Ord's metaphore of Gauss' Euclidean axioms, ROC relaxes the "axiom of code execution" and allows the foundational informational fabric of identity and context to take their place as first order citizens.

NetKernel is an embodiment of an ROC computation environment and it is, of course, ultimately represented as an inscription on a Turing tape (you can't escape technological reality). However a big trick that we pull with NetKernel is to have figured out that you can use knowledge of the thermodynamics and the statistical distributions of real world problems to trade the impossibly-infinite spacial extent of the Hypercomputation engines, for a local relativistic knowledge of resolvable context, computational energy and relative entropy of the representational state.

¹² <http://aturingmachine.com>

If you really dig down into how NetKernel is working you'll find that it is like a hybrid emulation of the "Oracle-machine" (Turing's own 1936 two-phase computational model), a "Turing Machine with Initial Inscriptions" and "Coupled Turing Machines". I say "emulation" since by necessity it is confined within a T-machine. However we can make it work since we are bounding the infinite state space requirements of Hypercomputers by trading space with energy and time to achieve a dynamic-equilibrium.

You tell me if NetKernel is the "calculus" step from the "algebra" of today's classical software? But I do know that NetKernel allows you to jump off any given Turing tape, peak at the dynamic partial multi-dimensional "Oracle tape" and jump back onto another, different dialect, Turing tape. Each time you jump off the tape you get the opportunity to allow the identity and contextuality axioms of the Platonic information reality to join the abstraction.

By making software the art of defining context, ROC is able to present solutions that are not bound within the uni-dimensional solution space of a single language. So you can approach a local approximation to a normalized representation of the software solution. Which is a long winded way of explaining why people repeatedly tell us "I can't believe how little code I write on NetKernel".

If this made sense you may also be starting to see how we can justify the claim that ROC allows architecture and code to be treated independently. Architecture is just another way of saying structural context (the domain of phase one of ROC). Code is the classical "algebra" (the domain of phase two). Ultimately this is important to us all since the entanglement of architecture with code is the limiting factor on the economics of realworld solutions. As I've said before: classical software is too damned expensive, too damned brittle and too damned small-scale for the information problems we could be tackling.

In summary, the heretical idea behind ROC/NetKernel is that software is not about the "algebra" of languages running on Turing machines, its about the "calculus" of obtaining information (state) by controlling and manipulating context.

Besides which, changing job title from "Software Developer" to "Director of Context" sounds so much cooler. ♠

How do you prove the value of ROC?

Peter Rodgers, PhD

Last week I got hot and bothered about linked resources. This really was a symptom of my wider issue of how to convey the meaning and value of ROC to a world that is only in the early stages of embracing the subset of ROC which REST embodies.

So what is the value of ROC? The easy things to point out are the first order facets: like performance, scaling etc. The recent scaling analysis and nk-perf tools we published provide some concrete evicence of how ROC yields first-order benefits, being impedance matched to multi-core processing architectures and the way ROC caching provides systemmic extrinsic memoization...

<http://www.1060research.com/netkernel/scaling/>

All very nice. But to focus there is to almost entirely miss the point - since with an underperforming conventional system you can always throw more compute power at problems. So performance is nice, but not sufficient of itself.

To explain where I think the killer value of ROC is, I need to tell you a short story. Back in the mid-90's I was working in HP Labs. I'd joined HP straight after finishing my PhD in quantum mechanics and had had a lot of fun dipping my toe into all sorts of technology areas. I'd come to realise I wanted to really dig into researching the Web as the basis for software systems beyond just, what at the time was, browsers and HTML content.

Because I'm lazy, the thing that appealed to me was that the Web was (still is!) an address space of potential reifiable state. And that was home territory, since that is very very similar to quantum mechanics: until you make a measurement all you have is a (Hilbert-) space of potential state. Measurement causes the collapse of the infinite potential state to a concrete information value. Pretty much exactly what happens when you resolve a resource identifier of an abstract Web resource to a representation value. REST was a no-brainer (or it would have been if there was a word for it back then).

So having cockily thought 'I "really get the Web" now to push it to the limits'. I wanted to build Web-scale information architectures that could tap these insights into the conceptual equivalence between the Web and QM. But to do that I could only build stuff with the software technologies that were available at the time, J2EE, EJB, Servlets etc etc. I got the shock

of my life. It was terrible. The intricacies of language and higher order APIs meant that you just drowned in irrelevant detail. That or the effort to build even relatively simple systems just didn't justify the cost.

Someone in the real world with a business imperative and a deadline would have pushed on, maybe created a higher order abstraction to solve for the problem domain. Maybe even have gone down the Spring route of stepping away from J2EE to an API model that was actually useable.

But being in a research environment I had the very rare luxury of being able to step back and consider the situation. What I'd hit was not a technical problem. I'd hit an economic one. The things I wanted to do just weren't economically viable. Another factor that was on the horizon ruled out building higher and higher order abstracted object models. Flexible data formats were starting to emerge, especially in the, then, brave new world of XML - look at the number of vertical industry domain working groups creating schemas left, right and center in the late 90s. This effort was all very worthy, but then you realise that each dialect needs a software embodiment and you are looking at the birth of the "million bastard offspring of EDI". Even worse, if you could afford to club-together and build an implementation (for example Rossetta net - the technology industry's supply-chain and procurement standard and implementation system), almost sure was that you'd never be able to afford to change it. What good are a million protocols that are locked in the late 1990s because the economics of embodying them in software dominate.

(I'm nearly at the original point about the true value of ROC! Honest.)

Armed with this perspective, we started thinking about the fundamentals of software. Why did the Web succeed? How could we bottle the essence of it and apply it more generally - both in the large and the small scale?

I updated our NetKernel page this week. I tried to succinctly answer the question "How do you prove the value of ROC?" see...

<http://www.1060research.com/netkernel/>

Here's a direct quote...

"Strangely perhaps, but less so when you've played with NetKernel and explored some of the concepts behind ROC, our most compelling evidence is the World Wide Web; the most successful software system ever created. The Web succeeds because of its fundamental economic property: All information systems are compelled to change. For the Web, the value added by a change is greater than its cost. More concisely, the Web is a software model in which change is cheap.

You might not have realized it but the Web is an ROC system. We demonstrate this in the first NetKernel tutorial, where it shows that the Web is an instance of just one basic ROC pattern using a single ROC address space.

Its cheap to create, modify, augment and evolve solutions in the Web. The Web is an instance of an ROC system, it follows that it is cheap to create, modify, augment and evolve solutions in ROC. The existence and success of the Web demonstrates this over the long term. NetKernel is a generalization of the ideas behind the Web, for the first time it allows the Web's economic values to be applied to any class of software problem both Web and non-Web."

Maybe I should have updated the "Are you crazy?" section with "Yes".

Maybe, maybe not. Actually, and here's the spin, Randy told me about a conversation he had yesterday with an established customer. They have an NK production integration solution that glues together mainframes with workgroups with you name it. One guy maintains it. While the rest of the organization is scheduling a month or more to implement new features, when a change request comes to our NK customer he can implement the features in a day or so. The rest of the organization still hasn't caught on to his secret so he is the super-hero in the group. ♠

On Empiricism

Peter Rodgers, PhD



Once upon a time there was a small town in a land not that far away...

The people of the town were very clever. They were renowned far and wide for their craftsmanship.

Everything in the town was clockwork. Everything in the town ran like clockwork.

The buses and trams were clockwork. The houses were powered by clockwork. The kitchens had clockwork washing machines and clockwork cookers. The factories made clockwork mechanisms. The schools taught the town's children the rules of clockwork.

The people of the town loved living there and over the years the town had become moderately wealthy. As an expression of their civic pride and as a demonstration of their skills, the people of the town decided to make a clockwork Mayor.

The very best craftsmen were selected and, after many years work, they proudly unveiled a full size walking, talking clockwork Mayor.

The Mayor was wise and beneficent and the people happily embraced his simple rules and the well ordered lifestyle.

After a year in office, things were progressing nicely. Everything was running like clockwork.

But there were still some aspects of life that had not been updated to clockwork. The baker's oven was still powered by logs. The cows were still milked by hand. The chimney's were still swept by hand using brushes. And of course the clockwork would wear down and need maintenance and even, as progress was made, would need replacing with new clockworks.

The Mayor saw these things and decided now was the time to set out a step-by-step plan for the town...

"We have made great progress with our expertise and mastery of the spring, cog and gear; but we can go further.

Within the next ten years we shall have reached a level of skill in which no aspect of our lives cannot be enhanced through clockwork.

Indeed, I proclaim that one day we shall have the ability for clockwork to design and produce clockwork. The days of tedious labour, filing gears and tensioning springs, will be past - we shall simply specify our requirements and the clockwork will be conjured for us."

The people rejoiced. This was a grand vision. A world of Complete Clockwork.

And so they went about their lives happy that the clockwork future was limitless...

Little did they know that the future was to be short-lived. For, entering the first year intake at the school was a small spectacled boy with shy demeanour but a prodigious talent for clockwork.

The small boy attended lessons, assiduously completed his homework assignments and progressed rapidly. As the years in school went by, he was taught about, and began to contemplate, the Mayor's vision for Complete Clockwork. The boy was uneasy - he couldn't quite say why, and, not being able to discuss this with the happy town's folk, he held his tongue.

Eventually the boy entered his final year and, as was tradition at the school, the children were required to produce a project demonstrating their clockwork skills. The small boy stowed himself away in his bedroom workshop to work on his project.

Days, weeks and months passed.

Eventually, one slightly grey morning, the boy placed his project, covered by a sheet, onto a clockwork barrow and headed off to the school for the assessment ceremony.

The Mayor was present, along with the proud families of the graduating students. Each pupil took it in turn to demonstrate their work and in turn would receive the praise and applause of their community in the form of enthusiastic chanting of "tick-tock, tick-tock, whirrrrrrrrrrr". The final 'R' being rolled in the manner of a Scottish dialect.



Finally it was the boy's turn. He tentatively pulled away the sheet. The people gasped - they had never before seen such a beautiful and elaborate mechanism. Simultaneously simple and yet apparently endlessly complex.

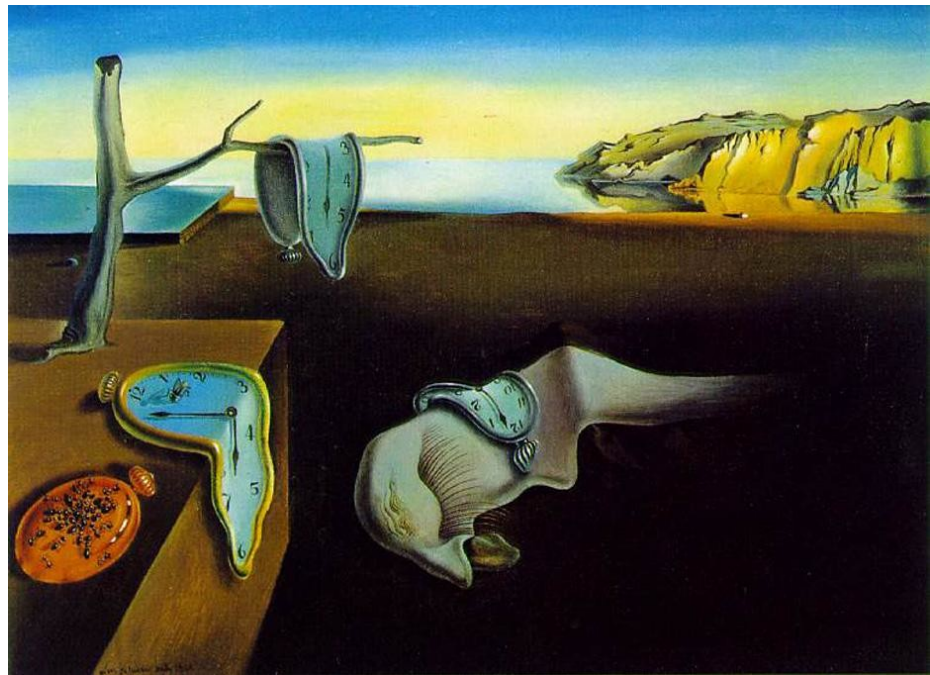
The boy said, "This is my Complete Clockwork Engine". Another hushed gasp. The Mayor beamed a clockwork smile.

With a flourish, the boy pulled away another sheet, one, it now appeared, of several on the barrow. He announced, "Here is a clockwork weaving machine which it built for me. The sheets were made with it". The crowd cheered.

The boy said, "But, there is a problem". The crowd stirred uneasily. The Mayor's smile faded.

"I have, on endless occasions, asked it to create me another Complete Clockwork Engine.", the crowd frowned, it needed some effort to think about this, for it was a brand new idea to them.

"Each time I have tried, the Complete Clockwork Engine has jammed. At first I thought this was my lowly skills. But I have come to understand that the Complete Clockwork Engine is not Complete. There are clockworks it cannot make. Clockworks that cannot exist.". The crowd gasped, the Mayor wobbled imperceptibly.



The boy was very clever, he had realised that he needed to prove his insight. He quickly snatched away a third sheet covering a very small clockwork.

"This is a clockwork proof. This machine will count springs. But no matter how you start this machine it will never stop. The cogs and gears will wear away before it will give an answer."

Finally, while the crowd were still reeling, he said, "This is my second clockwork proof.", and revealed an even smaller machine, and this one was very simple indeed. A series of in-

terlinked cogs each progressively larger than the last. "You can see that it shows that no matter how well you make your cogs, there is a point beyond which the complexity and interplay of the parts will always lead to a catastrophic jam up."

The Mayor was incandescent with rage. This could not be. This must not be.

But the people had intimate knowledge of clockwork - they could see that the machines embodied a deep truth. The world could never be the same again...

This was my thinly veiled parable, the Parable of the Clockwork Mayor. It was an attempt to present a reflected view of a real historical progression. *The progression away from Empiricism*. A convulsion which the intellectual world was shocked into making during the first half of the twentieth century.

Ultimately, I want this to provide a context in which to consider our world of Information Technology. But, as you will see, its vital for our conversation to have a general sense of the Progression from Empiricism.

This is my, admittedly, personalized perspective...

17th and 18th century

Empiricism really got going at the turn of the 17th and 18th centuries, although the ground work was laid a century earlier with the Copernican revolution, Galileo, and the advance of Science as a formalized process of exploring and rationalising our interactions with the world.



Towards the end of the 17th century it was beginning to seem like every area of endeavour could be rationalised and, with enough understanding of the process, be described by a prescriptive set of rules.

The world appeared to run like clockwork.

Even the stars were uniform, predictable, and it was a booming business to build elaborate clockwork orrery, in order, literally, to encapsulate the workings of the universe in clockwork.

The icing on the Empirical cake was delivered by Isaac Newton.

Newton is best known for his unifying explanation of the motion of the planets with the discovery and formal description of the Theory of Gravity. Today, we kind of take this for granted, but this idea was brand new - it unified, simplified and revolutionised everything. It was shockingly elegant.

It was also beautifully in step with the Empirical description of the world.

Here was a set of equations upon which it appeared the whole world revolved. Fire a canonball at such and such a speed, and such and such an angle, and it will land at such and such a point. (Ballistics and modern warfare got one hell of a kickstart from Newton).



There are great men, and there are Great Men. Newton goes in the second bracket.

People don't generally appreciate that Newton was not a one trick pony. He didn't just smash apart our world view with gravity. He did it twice. He developed and presented the first Theory of Light and Optics.

Before Newton, people didn't really know what light was. For example, it was an acceptable belief that it was something which was emitted from the eye.

Newton showed that light is a wave. That it can be separated into its spectrum and that individual colours are indivisible. He showed that light waves would interfere to make predictable and precise patterns (like ripples on a pond). He showed how light could be harnessed and turned into a precision tool.

Newton was adding cherries to the icing on the Empirical cake.

Light, gravity (not to mention he and Liebnitz's discovery of calculus) were obviously the precise, predictable, orderly rules. Everything else followed and any problem could be solved by a suitable mechanised approach.

19th and turn of the 20th century

If the 18th century was marked by the discovery and refinement of the empirical rules that underpinned the universe, the 19th century showed the consequences that must inevitably follow...

Water powered mills, gave way to steam powered mills which led to industrial manufacture, which led to unimaginable acceleration of productivity, which led to wealth, which funded empires, which changed every corner of the world.

The industrial revolution was the triumph of Empiricism.

Machines of manufacture were precision empirical engines. Anything that could be made, could be made by machine. It was entirely obvious that the world and everything in it followed predictable, mechanical rules.

Now, step forward to the turn of the 20th Century...

The outstanding intellectual force of the age was [David Hilbert](#). The German mathematician. Anyone who was anyone looked up to Hilbert as the definitive mind of his generation.

Hilbert, was surely a genius. In a stunning proclamation to mark the start of the new century, he set out a triumphant set of challenges to the mathematicians of the 20th Century. Today, some of these challenges have been met, but as a testament to his vision, many still remain to be resolved.

In the first article, I was being historically cruel to Hilbert. I set up the clockwork mayor in a way so as to tease him.

The reason being, that, nestling innocently amongst his challenges, was the mother of all Empirical visions.



Hilbert conjectured that mathematics itself would yield to the inevitable march of Empiricism. He proposed that the formal methods of deductive mathematical reasoning could be implemented as an empirical process. In short, that mathematics would be mechanised.

I should not have been so cruel. For Hilbert was sitting atop the pinnacle of human achievement that was the 19th century. We cannot blame him and his peers for wanting more of the same. He was reflecting everyone's world view: "The world is clockwork. Soon every secret of the universe will be laid bare and nothing will be beyond our reach."

The idea that mathematics (the basis of all the sciences) was a closed empirical world was immensely powerful and attractive. That there were no holes, that everything that could be expressed could be proven (or disproven) was the ultimate triumph of Empiricism. It came to be known as "Completeness". A complete formal process that underpinned everything.

Complete clockwork.

Things Start to Get Weird

It would be historically convenient to say that things then started to get weird.

No doubt you can anticipate some of the story that follows, but, in fact, before Hilbert's proclamation, things had already started to get weird.

In the 1870's Cantor gave the mathematical world its first dose of serious weirdness with his introduction of Set Theory.

Today we learn about Sets in junior school and we don't really get the point. "What use is a stupid Venn Diagram", we think.

The reason that we are introduced to sets is that we now understand that they provide the axiomatic basis for mathematics, even for numbers themselves.

When Cantor introduced the 19th Century mathematical community to Sets, all they saw were problems. Huge, nasty, frightening problems.

You see its easy to think about a set of two apples, or a set of three balls. But things start to go weird when you say "The set of whole numbers".

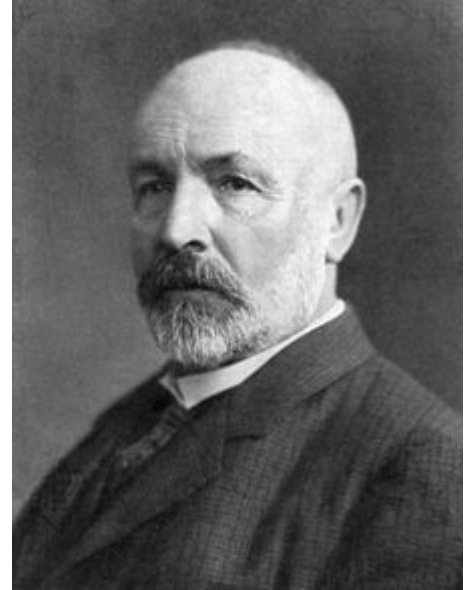
Things go really weird when you consider the "set of all things that are not apples".

If we think of this physically then we can sort of get our heads round it. But if we think of it strictly mathematically, the "set of all things that are not apples" must surely have itself as a member since it is a mathematical entity that is not an apple. But how can a set include itself?

However, things get really really weird when you start to try to classify how "big" a set is - its cardinality. How big is the set of Integers? Well its infinite right? What about the set of all rational numbers? Also infinite right? So they're the same size, yes? No they are not.

It turns out, and this was so weird it provoked a very strong reaction, there are different sizes (or rather "degrees") of infinity.

But this wasn't the only weirdness.



When you are required to consider infinitely large sets you start to discover that there are statements that are simultaneously true and false. We call these Paradoxes.

The more you consider sets the more you're likely to stumble into a paradox.

We all know about Bertrand Russell's [Barber Paradox](#) - the barber in the town shaves all the men who do not shave themselves. Who shaves the barber? (This is a variation of the paradox of the "set of non-apples" containing itself). [Richard's paradox](#) is worth a look too, since, relevant to ROC, it is a paradox of identity and representation.

The mother of all set theory paradoxes was an infinitely large hole that Cantor fell into - [Cantor's paradox](#), the gist of which is that there are an infinitely large number of infinities!

Health Warning: *Take care with set theory and infinity. Cantor eventually lost his mind. There's a recursive limit to our brains that you start to sense when you think too hard about this stuff. It bodes well to step back from the edge when you feel you're getting too close.*

To his credit, and by way of apology for my assassination attempt, Hilbert was a strong defender and advocate of Cantor and Set Theory. However its inherent weirdness didn't prevent Hilbert wanting to lead the Empirical charge into the 20th Century. "Assuredly, the weirdness in Set Theory can only indicate that we have not found the formal empirical rules that must lie beneath...".

I gave a very brief history of the Triumph of Empiricism in the period from the 17th to the turn of the 20th centuries. Earlier, I told the Parable of the Clockwork Mayor.

Which means we're nearly ready to start thinking about information technology, but first we need to smash up the world...

Turn of the Century - the cracks start to grow

We have seen that by the end of the 19th century the bizarre consequences of Set Theory were already starting to place strain on the rigid structure of Empirical mathematics. But



the general faith in Empiricism was still firmly rooted.

As we have heard, David Hilbert set forth on an ambitious plan to construct a "formal system". A rigid empirical framework in which every mathematical statement could be expressed, and by a simple algorithmic process be mechanistically proven to be either true or false.

It wasn't just in Mathematics that Empiricism was riding high. In Physics there was complacency and a general belief that "we are close to knowing everything". A sense that the big things, like thermodynamics, electricity and magnetism, had been solved and the rest would follow merely by empirical handle-turning.

Boy were they in for a shock...

Just after the turn of the century, almost simultaneously, two related but separate phenomena of light were observed. The most dramatic for many was the so-called [Ultraviolet catastrophe](#) in which the observed spectrum of black-body radiation diverged from the expected thermodynamic model at high frequencies. The second was the discovery that under certain conditions light could stimulate the emission of "electricity" from metals - the [photoelectric effect](#).

Let's digress a moment and explain why Einstein is a Great Man ie like Newton before him, he was more than a one trick pony: Einstein explained the photoelectric effect and in so doing was the very first to conceive the idea of quantisation. It is often forgotten that before Relativity Theory, Einstein was the godfather of quantum mechanics and indeed it was for the Photoelectric effect that he won the Nobel prize.

It was ultimately down to Planck and Einstein respectively to resolve these unexplainable phenomena. In both cases the explanation demanded that the energy and emissions were "lumpy". That is, the only explanation that worked was to recognise that the systems were not smooth and continuous but quantized into discrete states.

The Empirical Dam Bursts

The shocking aspect of these new theories was that they hinted that the world of continuous smooth empirical clockwork seemed to have discrete limits. As with set theory before them, the classical world was starting to discover that things were a lot weirder than was comfortable.



Motivated by the mounting experimental evidence that no-matter where you measured it, the speed of light always seemed to be exactly the same, even if you were on different sides of the earth/sun spinning at thousands of miles per hour difference, Einstein stuck it to the Empirical world big-style by asking a simple question:

"What would you experience if you travelled on a beam of light?"

The answer turned out to be almost incredible. For if the speed of light is universally constant, then the only possible answer is that space and time must bend.

One hundred years later, through repeated exposure to it and repeated experimental validation, we're kind of getting familiar with this idea. But I don't think the deeper significance is common currency. That space and time bend is one thing, but relativity theory states something even more profound:

Every observer experiences a different reality. Reality is relative to context.

That's still incredibly shocking. But the shocks kept on coming...

Collectively the community of researchers exploring the microscopic phenomena of quantization began to establish the experimental and mathematical tools for describing these effects and waved goodbye to Empiricism forever.

Quantum Mechanics showed the world to be inherently discrete, but incredibly, any given state could never be measured with precision. Precision could only be achieved either collectively (macroscopic ensembles) or by compromising the precision of another property of a system.

Worse still for Empiricism, the only way to explain the experimental evidence was to accept that the very nature of physical reality was itself intrinsically probabilistic. That is, stuff is not concrete at all, everything is a smeared out set of possibilities!

Put this together and you have the disconcerting truth:

Reality is contextual to the observer, it is non-deterministic and the precision with which we can interact with it is necessarily finite.

An empiricist would be weeping into their beer at this point, but in fact, the lifting of the rigid hard boundaries at the base of our scientific description of the world has led to an explosion in technological opportunities that even after half a century of mining its treasures, we are still only beginning to exploit. Your smartphone being just the tiniest tip of the iceberg.

But we're still some way off the information technology I want to address, so lets return to mathematics...

Hilbert's Cogs Slip

While the earth was shifting beneath the feet of the scientific community the Mathematical world could hold on to Hilbert's manifesto.

Indeed the first decade of the 20th Century saw a great triumph as Bertrand Russell and Alfred Whitehead published the [Principia Mathematica](#). Using set theory to provide a formal description of number and, as a crowning achievement after two huge volumes, to provide a proof that one and one makes two.

Very soon afterwards (well it was twenty years, but in Mathematics that's like the next day), up pops a precocious genius - the kid in the clockwork mayor story. His name is Kurt Gödel. Twenty five years old from Vienna.

He wheels his clockwork creations into the center of the mathematical community and shoots the mayor right between the eyes.

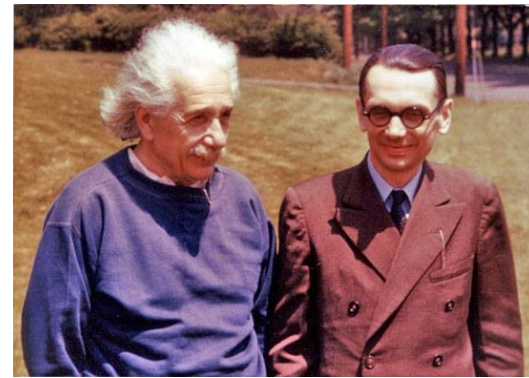
On his wheel barrow he had not one but two theorems: [Gödel's Incompleteness Theorems](#). The summary of which is that Hilbert's dream of a formal system can never be realised. Given any set of mathematical axioms there exist statements in the axioms that cannot be proved. Mathematics is incomplete.

Now lets get a sense of the impact of this.

Mathematics, like software, is a great place for the megalomaniac, since, unlike physical reality, you are only constrained by your imagination and your ability. If you run out of road with one set of axioms you can always choose new ones. But the implication of Gödel is that no matter which formal system you choose - there are statements you will not be able to prove.

Mathematics can never be empirical. As with the relativistic and probabilistic "unknowable" underpinnings of the physical sciences, Mathematics is also ultimately "unknowable", or rather more correctly, "incomplete".

Shocked? You should be. Our last tether to rational empirical reasoning just snapped. Its not elephants all the way down. We can't even know if there are elephants, we can't even



Einstein and Gödel, two of the greatest vandals of all time.

know if there's a "down".

On that bombshell we shall pause. Next time I promise we will get to the IT and I hope synthesize some valuable perspective from this mess...

The early Twentieth Century was a period of cataclysmic shocks. A brutal, unrelenting assault that left the Age of Empiricism in tatters... or did it?

In this part we shall finally move our attention to Information Technology and contemplate how the death of Empiricism came to be overlooked...

Modern Computing is Conceived

Modern Computing was conceived in the late 1920's and early 1930's.

History is a fickle discipline and we often simplify to make it easier to tell a story. It would be easy to say that "Computing was invented by Alan Turing" (I know you've been waiting for him to turn up since part 1). In fact modern computing and the formal mathematical concept of *computability* was conceived independently and approximately simultaneously by both Alan Turing and Alonzo Church.

The former is famous because his conceptual model of the Turing Machine is extremely simple to grasp, and ultimately, is a small step away from how a physical computer actually works. I suppose he earns the credit as the "Father of Computing" because he stuck around after the conception and actually helped deliver the baby and changed its nappies (diapers).

In parallel, Church devised the lambda-calculus, a much more "Mathsy" approach but, as is well understood, is exactly equivalent in terms of its expressiveness, to a Turing machine.

So, Computability has two fathers, but what does it actually mean?

The defining statement of computing is the [Church-Turing Thesis](#), which I shall express as:

Any calculation that can be performed by a human-being following an algorithmic procedure can be performed by a Turing Machine.

Sometimes this is simplistically, and wrongly, expressed as "anything that can be calculated, can be calculated by a Turing Machine", but as we've learned from Gödel, this massively over-eggs the pudding.

At this point we can probably fold-up the intervening 80-years and move straight up to the modern day. After all, that's still what we do in IT now isn't it? We spend our days "constructing algorithmic procedures that can be performed by Turing Machines"...

"What are you doing?"

"I am constructing an algorithmic procedure that can be performed by a Turing Machine"

...is a long winded thing to tell your manager, so we generally shorten this to "coding". We say, "*I am writing code*".

It seems like Empiricism didn't die after all? Could it be that Coding is to Empiricism what the Birds are to Dinosaurs?

It certainly feels like our building materials (languages - imperative, functional; procedural, object oriented) and our working practices - test driven development - would wish this to be so?

Modern computing has been a great success. Its all pervasive. Its a multi-billion dollar global activity. It feels a lot like the empirical age at the end of the 19th century.

If history tells us anything then IT is ripe for an empirical catastrophe.

Or it would be, if IT's catastrophe hadn't already happened (twice) and we didn't pay attention...

Computing's Empirical Catastrophy

We have heard that Gödel pulled down the pillars of mathematics. His incompleteness theorem was a devastating shock in mathematics - the shock was large in direct proportion to the millenial-depth of the mathematical foundations it undermined.

The weird thing is Computation has had just as serious a shock. Its just that Computation's empirical catastrophe came while computing was, as yet, unborn. Even weirder, the creator of computing, Alan Turing, was also its destroyer...

Turing discovered that he could easily construct algorithms (encodings on the Turing Tape) that would result in the Turing Machine running forever. If you were a child of the 70/80's no doubt you too also quickly discovered this BASIC truth...

```
10 GOTO 10
```

Unlike you or I, playing with our first programs, Turing put some mathematical rigour into this discovery. In doing so he joined Gödel as a destroyer of worlds (*Just as with buses, you wait millenia for an incompleteness theorem to come along and then within a matter of years two come along at once.*).

Turing proved that there is an infinitely large set of programs (Turing Machine encodings /algorithmic procedures) for which it is impossible to prove in advance if they will end (halt). In Computing this is called the Halting Problem and it is, in fact, exactly mathematically equivalent to Gödel's incompleteness theorem. Turing knew this, his mathematician peers knew it; put simply...

Computing is incomplete.

Shocked? No, I thought not. It was a long time ago. Before the first computer had even been built. Before programming languages, operating systems, microprocessors... We seem to rub along just fine. Who cares?

The definition of computing lets us happily stick to our professions "constructing an algorithmic procedure that can be performed by a Turing Machine". Never mind that the Halting Problem tells us that there is an innate limit to the complexity of any empirical procedure we can devise. We can sense this by recognising that the ceiling of complexity necessarily limits us to "those algorithms which run in short enough time that we can verify that they halt".

Computing's Second Catastrophy

Smash my world once and I don't notice, more fool you, smash my world twice and I still don't notice, more fool me...

We might be forgiven by our Mathematical colleagues for not being moved to tears by the Halting Problem. It was all a long time ago, before our discipline had even been born. So how do we reconcile the catastrophic events of the 1960's? Step forward Gregory Chaiten and Andrei Kolmogorov...

By the 1960's computing was really taking off. It was proving really quite useful and there was money to be made: the CEO of IBM predicted the world might need at least a hundred computers. The first generation of programming languages were maturing. Things looked great - we were all set for the birth of operating systems and the microprocessor revolution - the forward march to the future was on an unstoppable track...

But those pesky mathematicians start asking difficult questions again. Questions like, "what's the shortest possible program to compute any given problem?", which begs a harder question, "How do we determine how complex any program is?".

Kolmogorov answered this last question by defining a measurement of the [complexity of an algorithm](#) based upon its string encoding and showing how any minimum representation of a complex problem must have the same entropy as a random representation - if it wasn't completely random then there must exist a smaller algorithmic representation able to represent it. This follows by the simple observation that what "non-randomness" really means is that "there is a recipe to generate" something. *(You see I'm not the only one to bang on about entropy in representations).*

Meanwhile, a little like the precocious Gödel a generation earlier, Gregory Chaitin a kid of twenty-something wheeled his own clockwork wheelbarrow out and blew a great big hole in the fabric of computing...

By taking a similar approach to Gödel and exploiting the Berry paradox of set-theory, Chaitin presented a new [incompleteness theorem](#), which in its mathematical form is pretty hard to understand:

For every formalized theory of arithmetic there is a finite constant c such that the theory in question cannot prove any particular number to have Kolmogorov complexity larger than c .

Generally, and somewhat [controversially](#), this is interpreted as: you can only prove something of a given complexity if your axioms have at least the same complexity.

Shocked? No, I thought not.

But you should be. You should be really really shocked. Irrespective of the detailed interpretation, there are direct and immediate corollaries we should care about profoundly:

For a problem of a given complexity, you cannot know if you've discovered the shortest program to solve it.

And it follows...

You can never know if the language you are using is the most elegant for a given problem

And it follows...

There are no special programming languages: there are only problems for which they "seem to fit" and equally problems for which "they don't".

You cannot predict before hand if you're using the "best language". To a fundamental degree, based-upon first principals: software is a faith-based discipline.

On that bombshell we shall pause. Next time, we'll consider where that leaves test driven development and what happens if we reconsider our definition of computability, what happens if we, as with our scientific peers, relax Empiricism and admit relativity and probabalistic approximation... ♠

NetKernel 5

The World's 1st Uniform Resource Engine

ROC Microkernel
Heatmap
NK Protocol / Cloud Infrastructure
Visualizer Time-machine debugger
Resource Resolution and Classloader Trade Tools
L2 Cache
Load Balancer
Endpoint Statistical Profiling
Encryption Modules
Real-time System Status Reports and Charts
Apposite Package Manager
xUnit Test Framework
Multit-mode deadlock detector
Role-based System Administration Security
DPML Declarative Request Language
Historical Log Analyzer
Multilevel Log Management
nCoDE Visual Composition Environment
eMail client / server
XMPP Transport
SOAP Transport
REST Web Services client / server
Relational Database Too Set
SSH daemon
Declarative Architectural Suite
oAuth client / server
High-performance Asynchronous HTTP client / server
HTTP Virtual Hosting Manager
State Machine Runtime
Groovy, Ruby, Python, Java, JavaScript, BeanShell

www.1060research.com