Abstract

Today's software systems have a much shorter life span than other engineering artifacts. Code is brittle and system architectures are rigid and not robust to change. With the Web an alternative model of computing has taken shape, a model that allows for change on a small scale inside of a complex and large-scale system. The core abstraction behind the Web can be applied to the design of software systems. The abstraction in Resource Oriented Computing generalizes the Web abstraction to enable the most adaptable software systems and architectures that evolve with change.

Resource Oriented Computing for Adaptive Systems

Peter J. Rodgers, PhD, info@1060research.com

May 2015

1 Challenge

The second Law of Thermodynamics states that entropy will always increase. The challenge faced by long-lived resource adaptive systems is to fight entropy with the least possible expenditure of energy (physical, computational, financial).

2 Definitions

"Myriad computational systems, both hardware and software, are organized as state-transition systems. Such a system evolves over time (or, computes) by continually changing state in response to one or more discrete stimuli (typically termed "inputs"). When in a "stable" situation, the system is in a well-defined one of its (finitely or infinitely many) states. At any such moment, in response to any valid stimulus, the system goes through some process, ending up in another "stable" situation, in some well-defined state." *A. Rosenberg*

It is helpful to start by stating some definitions so that we may clarify concepts such as "stability" and "evolvability" as applied to adaptive computation systems.

It will be shown that we can obtain a useful working understanding of *stability* and *evolvability* by considering computational state.[1]

We require to maintain a conceptual distinction between a given logical abstract potential state and an instantaneous physical representation of the state. We do this by introducing opaque identifiers such that any suitable resolution of an identifier will yield an adequate representation of the state of the abstract resource it identifies.

2.1 Resources

We must also qualify the overloaded term "resource". Historically *resource* has been used to mean the computational system, its physical embodiment, constraints and capabilities. In what follows we shall more frequently use the term resource to mean the *abstract potential state of a computational system* – in this respect we are using the term resource in the way that is now common with respect to the Web and RESTful systems. Where it is not possible by the context to infer this distinction we will use italics to stress that we are referring to the first historical form.

2.2 Stability and Evolvability

It follows that we may define *stability* and *evolvability* respectively:

Given identity **I** a system is *stable* under change if, when presented with **I** it yields a representation **R** of the state of a resource and, when changed such that **I** now yields **R'**, then **R** is still a sufficient subset of **R'**.

[1] "For an excellent treatment of the concept of "state" in computation theory see A. Rosenberg"s "The pillars of computation theory: state, encoding, nondeterminism".[3]

The system is *evolvable* if, in addition, as the identifier **I** is itself changed to **I'** then requesting **I'** also yields **R'** or another representation **R"** such that **R'** is still a sufficient subset of **R"**

2.3 The Web

By these definitions the Web (REST) is a *resource* adaptive system that is both *stable* and *evolvable*.

While the Web is an excellent exemplar of the properties we seek – it is not sufficient in itself as an architecture or as an abstraction.

Notably it relies upon a globally agreed restriction on the set of I – due to the simple flat address space provided by the DNS system. This restriction means that there is only an implicit conception of scope and so limited flexibility in the range and domain of validity of the set of identifiers, something that is essential in a computational system that addresses anything other than quite limited problems.

Equally, the Web is physically embodied as a networked application mediated by the HTTP protocol, which, while incredibly successful at macroscopic scale, is not suitable for the nano- second latencies one encounters in software at the microscopic scale.

3 New General Computing Abstraction

What is required is a general abstraction, one which extends and generalizes the properties inherent to the Web but which enables them to be applied to general computation.

3.1 Turing Tape

For the past seventy years we have accepted a model for software that inhabits a life "on the Turing Tape". Programs are encoded and execute on a Turing machine. Irrespective of programming language or its paradigm (OO, procedural, functional), there is a tight coupling between the code and the machine. We observe this in the universal acceptance of the code-compile-link-execute life-cycle of software.

It follows that any change to the machine (its resources, environment, operating context etc) require corresponding change in code in order to maintain a valid and stable computation (by definition above). Even when the execution of the machine is itself virtualized.

3.2 Physical Decoupling

The Web shows us that a computation model can exist that does not bind tightly to the execution environment of the Turing engine.

In the Web, each endpoint (computational unit) is linearly independent. Endpoints are not physically coupled (as in linked code) but rather are logically coupled via the resolution of resource identifiers (URIs).

It follows that a resource request in the Web may be satisfied by any sufficiently capable endpoint – irrespective of the encoding dialect, form or operating characteristics of the underlying Turing engine. Furthermore the computational demands of any given logical

resource may be satisfied by any suitably configured engine – it follows that the Web seamlessly scales from very small computational units up to almost indefinitely large ensembles of Turing engines. This property has become known by the term "Load-Balancing".

3.3 Research Experience

"Modern-day software systems, even those that presumably function correctly, have a useful and effective shelf life orders of magnitude less than other engineering artifacts." *(DARPA, April 2015)*

We first began considering the observations presented above in the late 1990's in the DEXTER research team at Hewlett Packard Laboratories. Indeed the challenges raised by the DARPA BRASS research programme are the same motivators that have driven the last 17 years of our RD efforts.

Our driving vision can be very simply stated: The engineering qualities of software are poor. In particular, classical software lacks the ability to adapt to changes in both the environmental execution context and operational platform. This is in stark contrast to the Web; an architecture which has demonstrated long term stability, continuous evolution and is robust in the face of both types of change.

3.4 ROC Abstraction and NetKernel Platform

Our research has led us to develop the *Resource Oriented Computing* (ROC) paradigm. ROC extrapolates the core elements of the Web model and generalizes them in order to provide a computational model that seamlessly accommodates change in both software and hardware.

We have developed a computing abstraction we call "Resource Oriented Computing". A mature implementation of the ROC abstraction is offered in the form of the **NetKernel ROC platform**.

The ROC abstraction generalizes the Web's REST abstraction so that it may be applied to general computation. The NetKernel embodiment exhibits the properties of stability, evolvability and scale invariance that we observe in the Web architecture.

It will become apparent that the ROC abstraction provides a uniform and complete model for arbitrary resource adaptive computation. In order to recognize the general potential of the ROC approach we must first show how it relates to the Web and how it generalizes beyond it.

4 Description of the Web Abstraction

In 1994, Tim Berners-Lee's proposals for HyperText implicitly introduced the idea of a URL as a short string representing a resource that is the target of a hyperlink. *(from Wikipedia)*

The essential elements of the Web abstraction are depicted in *Figure 1* below.

We may consider the global World-Wide-Web as a logical uniform address space. A participant (browser, software user-agent) may interact with the space by forming an identifier for a resource in the space. In the Web these are commonly URLs, a subclass of the general URI identifier structure.

In fact the address space of the Web is not physically real – there is no "Web Space". Rather the space logically exists as a consequence of the structure of the URL possessing a resolvable host name.

A participant in the Web must resolve the host name against the DNS system in order to locate a physical endpoint that will accept a request for the resource identified by the URL.

Having resolved a physical host, the participant constructs a request and furnishes it to the endpoint. In the Web this interaction is mediated by the HTTP protocol over a TCP/IP network.

An endpoint that is resolvable by this URL has committed to provide a representation of the abstract resource that is identified. It is of no-concern to the Web abstraction how the endpoint achieves this. In the early Web it was common for endpoints to implement static mapping to files. Today it is common for the endpoint to be implemented as some form of software container that will compute a representation of the state of the resource.

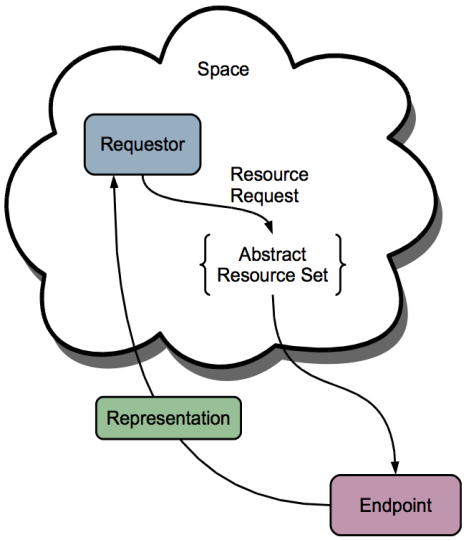

Figure 1: The Web Abstraction

4.1 Resolving Logical Abstract Resources

It is very important to emphasise that the detail of the implementation of the endpoint is entirely irrelevant to the Web abstraction. All that is necessary is that ultimately a Web endpoint must return a representation of the state of the logical abstract resource to the requestor.

It is understood from the REST architectural style [1] that it is desirable that an endpoint should be stateless and that any necessary state required to compute a requested resource should be transferred to the endpoint when it is requested – we shall see below that the principle of state transfer is very important in resource adaptive systems.

The Web is the most successful software architecture that has yet been devised. It has continuously evolved and adapted to change for more than twenty-five years. By being isolated from concern over the implementation detail of endpoints it exhibits total independence from the computation resource (physical platform, network topology etc) employed to deliver the system. Indeed the same logical system proves stable under continuous adaptation of the underlying physical compute resource.

4.2 Logical Information Resources

The innate resource adaptive property of the Web is readily illustrated by the simple observation that the resource http://www.google.com was at one point provided by a single endpoint on a single server - it is now provided by hundreds of millions of endpoints on tens of million servers. The essential logical information resource has remained uniformly consistent and stable and yet, the complete solution that is the “Google Application”, has simultaneously undergone continuous evolution (here we use resource, stable and evolvable by our earlier definitions).

5 Description of the ROC Abstraction

The Resource Oriented Computing abstraction derives from and generalizes the Web abstraction. Its objective, unlike the Web, is not to enable a networked application, but rather to provide a uniform abstraction for arbitrary computation – both local within a single physical compute resource and across networked cloud platforms.

5.1 Fundamentals

Before we can understand how general computational applications may be developed we must first describe the fundamentals of ROC. It is apparent that the essential elements of the Web discussed above are also present in the ROC abstraction.

A requestor desiring the state of a resource forms an identifier, the identifier is resolved in an address space to an endpoint that will accept the request. The detailed implementation of the endpoint is of no concern, what is required is that it must return a representation of the state of the logical resource to the requestor.

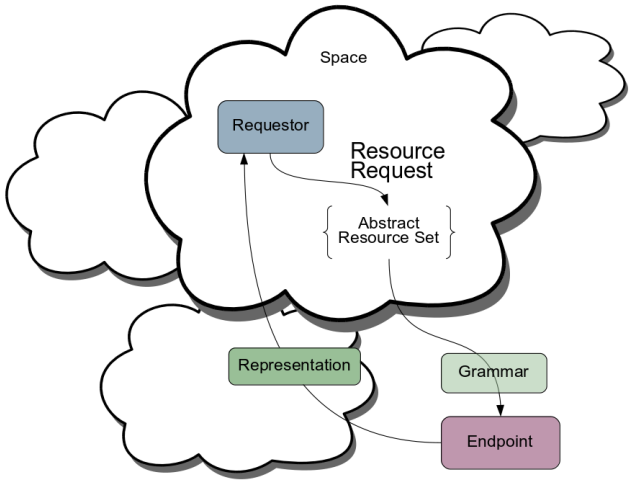

Figure 2: ROC Abstraction

The first way in which the ROC abstraction differs from the Web abstraction is in making explicit the nature and role of the address space.

In the ROC abstraction an address space (henceforth simply called a space) explicitly provides a means to resolve a logical resource identifier to a physical representation. This is not a semantic nicety – unlike the Web in which the space may only be inferred due to the nature of DNS and TCP/IP network, an ROC space is tangible and has an explicit role.

It is apparent that without the Internet as the medium (remember we are discussing a general computing abstraction not a networked application), we must consider the means by which a space should determine the resolution of identifier to representation.

Just as with an endpoint, the ROC abstraction is not prescriptive about the embodiment of a space, however our experience has shown that it is valuable to associate a grammar with an endpoint. We define a grammar as a pattern matching language such that when it is presented with an identifier for a resource it determines if the identifier should be accepted by the associated endpoint.

5.2 ROC Generalizes Web Abstraction

It is apparent that in the Web, the interaction between the user-agent, the URL, the host name part and the DNS to locate a physical host endpoint fulfils the role of a grammar.

Our experience of ROC solutions allows us to recognise that a grammar establishes a contract between a space and an endpoint. A grammar is a promise that if a specific resource, or more usually, set of resources is requested, then the given endpoint will be assigned responsibility to reify the representation state. The NetKernel platform provides several grammar languages, ranging from simple pattern matching through to expressive BNF-like syntaxes.

Typically a space will perform a linear search in order to resolve the resource identifier; presenting a resource identifier to each endpoint grammar in turn to determine a match. We will see below, that while non-linear resolution is not excluded, it is highly advantageous to perform the resolution in a linearly deterministic fashion.

It is apparent that the ROC abstraction generalizes away from the implicit special case of the Web and the physical heritage of its logical global address space. Additionally, and most significantly, the ROC abstraction makes a further extension beyond the Web.

5.2.1 Removing Limitations

In ROC, the limitation of, and adherence to, a single logical space is lifted. An ROC solution may consist of any number of spaces. We shall show below that the lifting of the single space constraint introduces the opportunity to formally determine and dynamically adapt operational context. Our experience has taught us that control and understanding of context is a critical requirement in any long-lived resource adaptive system.

5.3 Spacial Multivalency, Scope and Context

Without the limitation of a single space we are free to create higher-order architectures consisting of relationships between spaces.

The simplest mechanism to create a higher-order structure is to implement an endpoint, such that, when requested to resolve an identifier by a first space, rather than deploy a grammar it delegates the resolution to a second space.

Implementations of this pattern are collectively known as “imports”, since it is apparent that conceptually the requestor in the first space is able to resolve the resources of the second space just as though they were present in the first space. The resources have been “imported”.

It should be noted that this notion of import is unlike the classical model we have become used to with programming languages. An import is an endpoint and the indirection it provides to the second space may potentially be dynamic. We shall see below, that there are many possible embodiments

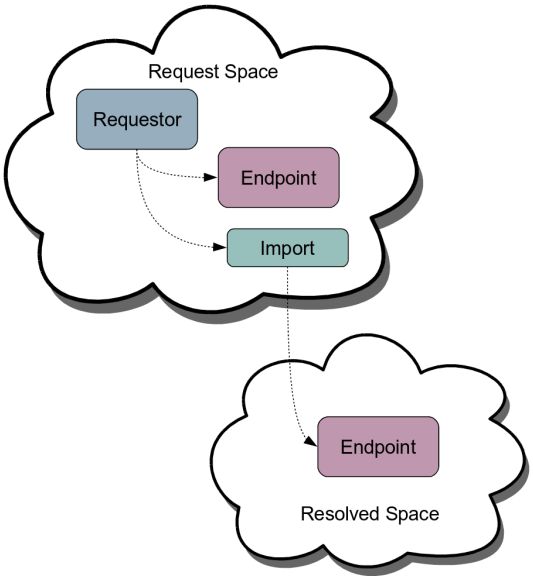

Figure 3: Import

of the import pattern and the routing performed by an import endpoint may be dynamically determined by the state of some resolvable resource, or, commonly, many resources.

The uniform treatment of imports in the resource resolution process leads to significant breakthroughs in the evolvability and stability of resource adaptive systems.

Returning to our consideration of context. It is apparent that the ROC delegation of resolution via imports, and the multivalency of spaces, enables an emergent determination of spacial locality. A request issued in one space, may be delegated through numerous spaces before it is ultimately resolved to an endpoint establishing an operational context for any given computation. Knowledge of the spacial structure determined during the process of resolution is extremely valuable and provides the scope in which subsequent resource requests may themselves be resolved.

5.3.1 Formal Context of Operation

The emergent dynamic discovery of spacial scope establishes a formal context of operation. However to appreciate the significance of this we must step away from our learned understanding of scope inherited from our experience of programming languages.

Here, we are not discussing the lexical or dynamic scope of variables or functions in a specific programming language coding model. We are describing a spacial structure in which potential logical resources may be requested. The form and function of the application to which the resources belong is therefore determined by the arrangement and structure of the spacial architecture that constitutes a solution.

To more clearly understand the potential of emergent spacial scope, it is useful to view a typical ROC resource adaptive application architecture. *Figure 4* (next page) shows a view of a dynamic emergent ROC application architecture. Shown at the center is an enlargement of the red highlighted region. At this scale it can be seen the entire solution consists of elemental ROC spaces housing resolvable endpoints.

5.3.2 Decoupled System

The system is entirely decoupled, in that, the lines shown connecting spaces are not static bindings but rather potential logical resolution routes established by various embodiments of the import pattern.

To more clearly understand the potential of emergent spacial scope, it is useful to view a typical ROC resource adaptive application architecture. Shown center left is an enlargement of the red highlighted region. At this scale it can be seen the entire solution consists of elemental ROC spaces housing resolvable endpoints.

Without hands-on experience of ROC, the architecture depicted in *Figure 5* appears to be very complex. Indeed what we are viewing is not the full story - we are seeing only a simplified slice through this multidimensional spacial architecture. The slice we have shown is the relevant application context for the left-hand most space (that space contains an external transport endpoint - in ROC such an arrangement is called the “fulcrum pattern”). If we were to view the full N- dimensional structure of this ROC system it would be an order of magnitude more complex again.

However it is important to emphasise that the observed complexity is emergent, and that this high- level perspective represents a broad view that results from the combination of many simple and orthogonal ROC patterns. Furthermore, an experienced ROC architect is

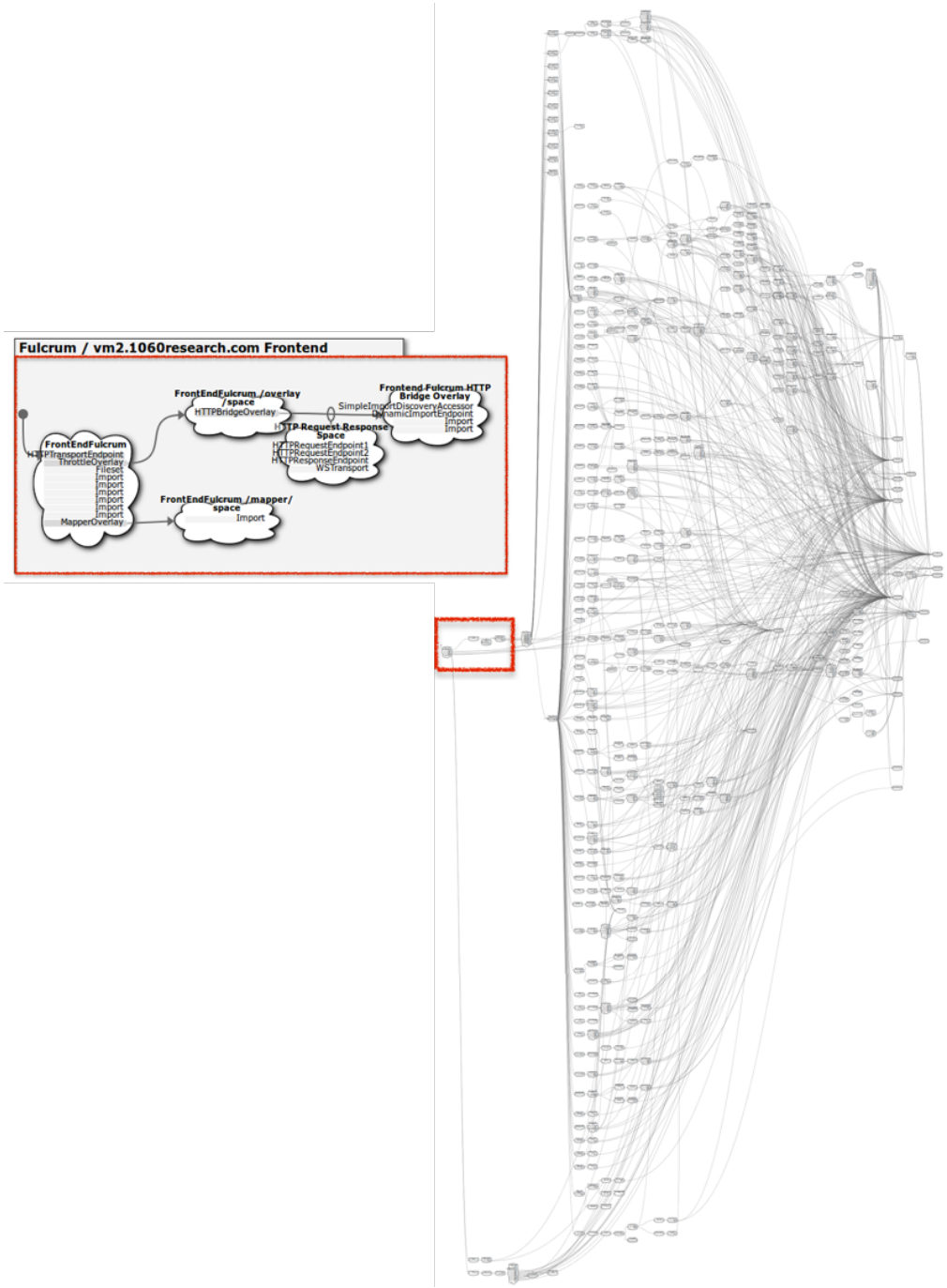

Figure 4: Snapshot view of a dynamic ROC application architecture taken with the NetKernel space explorer tool.

able to recognize and gain understanding of these systems simply by visually inspecting the high-level spacial structures.

Examining the example shown above, we might consider that the spaces shown are the result of rigid physically predetermined configuration. It is important to emphasise that a space is a lightweight construct and indeed it is a common pattern to construct ephemeral spaces on a per request basis. In the limit, it is in fact a capability of NetKernel to dynamically compute a complete ROC spacial structure, even on a per request basis.

5.3.3 Very High Reuse of Software

One final set of observations. It is important to recognise that in ROC solutions each individual space is linearly independent and so may simultaneously provide its services to many other architectural topologies – this property, combined with formal stability, results in software that exhibits very high reuse as evolution and adaptation is introduced.

Furthermore, since the architecture is logically defined and not physically linked, the structure may be dynamically modified. It follows that configurations of spacial architecture may be deployed or changed collectively without restarting a system (hot-deployment).

Lastly, it will be shown below that the dynamically coupled nature of ROC solutions allows specialized engineering constructs within an architecture to be dynamically adapted based upon the state of the underlying computational *resource*. This property results in software that is intrinsically resource adaptive.

We hope that, having established how ROC generalizes from the Web, it is apparent that ROC automatically inherits the Web's properties of *stability* and *evolvability*. These properties allow such complex systems to progressively *evolve* and adaptively respond to change.

5.4 The Web is Passive

We have necessarily taken some time to establish the core principles of the ROC abstraction. We are now in a position to consider how it is applied in practice and its relevance as a platform for *resource* adaptive solutions.

At this point one might reasonably ask, "So where is the code?"

We discussed earlier how in the Web, an endpoint is typically embodied as a black-box software container. So too in ROC. An endpoint within an ROC space is a software container. Its requirement is, when resolved, to receive a resource request and compute a suitable representation of the resource state.

From our experience of the Web, this does not at first appear to be a promising prospect. Our experience of the Web is influenced by its early history as a publication application. Indeed the primary tool we use to interact with the Web is still called a "Browser". Our experience of Web endpoints is that, while they may be implemented as dynamic software containers, they are "different to programs" in that they predominantly provide limited snapshots of the state of resources - often really just transformational views of back-end database state. Our expectation of the Web has become constrained by the "database-backed" Web-site paradigm.

Additionally, our expectation of an endpoint is further limited by our historical experience that software is an encoding that executes within the constraints of a specific underlying Turing engine. We live "on the tape" within a language and employ the classical code-compile-link-execute paradigm. Software solutions do not "step off the tape"?[2]

We shall now show that the Resource Oriented Computing abstraction permits us to break free from these constraints.

5.4.1 Dynamically Computed Resource State

An active resource oriented system should allow us to dynamically cause resources to interact – we desire a system in which it is possible for one resource to transform another and another and another... in a potentially indefinite cascade of dynamically computed resource state. In fact, the resource oriented abstraction permits exactly this dynamic active computational model. It does this by making one simple step – it admits that an endpoint is not simply an exit-ramp to a specific software container (Turing engine), it may just as well act as a requestor for resources, a symmetrical on-ramp back to the ROC abstraction (we saw hints of this earlier in the discussion of the import pattern).

It is apparent that, in order to satisfy an initiating request, a symmetrical endpoint can construct and issue further resources requests in order to use the requested resource state in the computation of the state of the initial resource request. We start to see that ROC is a world in which we can say "do this to that" or "do this to that with that and that" etc, etc. This is possible because ROC is a world in which stateless endpoints are able to acquire transferred state in order to perform arbitrary general computations. The desirable RESTful property of *state transfer* is preserved and generalized in the ROC abstraction. However, in order to express the rich interplay of an active computational resource model we must consider a suitable way to *identify* resources.

It is clear that the familiar URL is inadequate – its linear path-like structure betrays its publishing/file-oriented heritage. That said, it is clear that a singular triumph of the Web is that the URI is a stable, well understood, well formed unambiguous identifier structure. Therefore some form of URI would be a natural candidate for ROC.

5.4.2 Active URI

In 2004 1060 Research submitted a draft specification to the IETF for a new class of URI. One which would provide us with the ability to specify the identity of actively computed resources. We call this an "active URI". Endpoints and spaces in the NetKernel platform make extensive use of active URIs.

[2] An interesting theoretical discussion to extend the paradigm of the Turing tape as an "interactive Turing machines with advice" has been put forth by Leeuwen and Wiedermann. Their considerations are in several ways akin to the ROC implementation.[5] Moshovakis underscores the importance of such a conceptual overhaul: "The Church-Turing Thesis grounds proofs of undecidability and it is essential for the most important applications of logic. On the other hand, it cannot be argued seriously that Turing machines model faithfully all algorithms on the natural numbers. If, for example, we code the input n in binary (rather than unary) notation, then the time needed for the computation of f(n) can sometimes be considerably shortened; and if we let the machine use two tapes rather than one, then (in some cases) we may gain a quadratic speedup of the computation. This means that important aspects of the complexity of algorithms are not captured by Turing machines.[2]

For details we refer you to the IETF "active URI" draft specification, however a very brief introduction can be given with some examples...

```
active:service+resource@http://host/path
active:transform +operand@foo+operator@baa

active:language+code@some-code
active:groovy+operator@res:/script.gy+this@foo+that@baa+other@baz
```

The active URI consists of the "active:" scheme preceding a named service, followed by any number of name-identifier pairs united by an @ character (name@identifier), each pair is delimited by a + sign, each named identifier may itself be a URI.

An endpoint with a grammar that is capable of recognising and parsing the active URI can very easily construct requests for the named resources (the identifier in the name@identifier pair). It can be seen that the named resources in the active URI act as "resource arguments" to the endpoint. The state of the resources referenced in the arguments can be requested (transferred to the endpoint) in order for it perform is computation.

5.4.3 Language Runtime

One immediate consequence is that our constraint of being confined to a single Turing machine is lifted. We may implement address spaces in which resolvable Turing machines implementing execution environments for numerous coding languages are present (this pattern is called a "language runtime").

5.4.4 Code State Transfer

A runtime is requested using an active URI in which one named argument is the identifier of the resource that is the code we wish to have executed. The runtime issues a request for the code resource and then executes it. This pattern is called "code state transfer".

To complete the overview, the executing runtime code has access to the initiating request (and thereby its other arguments) and equally has the ability to request further resources etc. Hence code executing in a language runtime is fully capable of complex computations.

Figure 5 shows a depiction of the language runtime pattern and shows how the total computation is not local to a single Turing engine but may be distributed through the contextual address space.

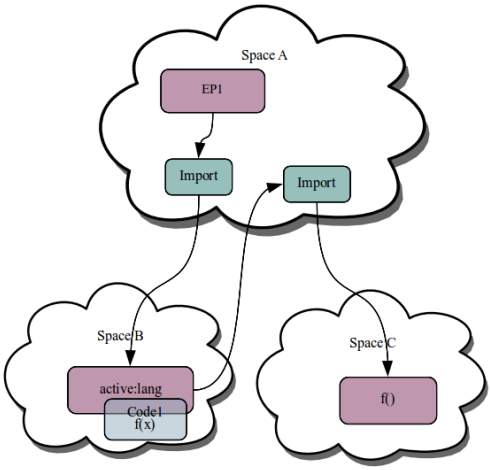

Figure 5: Language Runtime Pattern

The NetKernel platform, while natively written in Java, provides dozens of language runtimes that utilize this pattern. Languages

supported include Turing complete languages such as Groovy, Python, JavaScript, Ruby, Clojure, Java, XSLT, Xquery, DPML... together with several compositing languages useful for compositional operations on resources, as well as many domain specific languages, database query languages and more prosaic basic transformation tools.

Solutions are built by writing code, but it is apparent that code itself is a resource. Systems are adaptive and evolvable because the code that they execute is of the same form as the resources that are modelled in the computations performed by the system. In ROC everything is a resource.

Before we conclude, it is worth pointing out further significant properties of the active URI. It is multi-dimensional allowing complex resource models to be developed without being confined by a structural straight-jacket. Additionally, by expressing arguments by name rather than position, it is order independent. Lastly, it is recursive, in that any given active URI is also the identifier of a resource (every computation has an identity). It follows that an active URI may be a named argument of an active URI (provided suitable care is taken to escape delimiters). For example...

```
active:service+resource@active:transform%2Bfoo@baa
```

In NetKernel, the active URI provides a resource oriented "bytecode" and, if appropriate, extends the potential of the ROC abstraction to offer a functional programming model that is extrinsic to the Turing engine.

It is hoped that the significance of code state transfer to the problem of resource adaptive systems is recognized. The ROC abstraction allows that the entire computation of a given problem is dynamic and the program code itself is consistently modelled as resources within the abstraction. It follows that it is trivial to implement solutions in which code is a dynamically generated resource, which may be generated based upon the state of further resources including resources whose state depends upon the core underlying state of the computational platform resource. ROC presents a self-consistent platform for resource adaptive systems.

5.5 Application of the ROC Abstraction

What has been presented above has covered the foundational concepts of the ROC abstraction. Establishing the core elements of spaces, endpoints and resource requests. It was shown, in the language runtime discussion, how the simple elements of ROC combine to allow the conceptual understanding of how software systems are structured to be radically rethought.

It was shown that dynamic determination of spacial locality results in a contextual scope for execution. It can be seen that a language runtime operates within this dynamic contextual scope. As can be inferred from the example of a typical ROC architecture presented in *Figure 6*, this property leads to significant breakthroughs in architectural patterns, engineering constructs and the enhancement of the engineering qualities of a solution.

We do not have sufficient space to discuss in depth the wide range of consequences that lead directly from the ROC abstraction. Below we provide a small selection of the more important items including several examples of how resource adaptive analytics are available as part of the NetKernel tool set:

5.6 Systemic Memoization

In ROC all computation occurs by issuing requests for resources. It follows that the result of every computation has an identifier (and contextual scope). In addition, state transfer via the extrinsic cascade of sub-requests in the ROC domain means that the ROC abstraction does not admit side- effects.

These properties mean that the ROC system guarantees systemic referential integrity – that is, the state and dependencies of all computation is measurable and unambiguously understood.

Memoization is a way to lower a function's time cost in exchange for space cost; that is, memoized functions become optimized for speed in exchange for a higher use of computer memory space. The time/space "cost" of algorithms has a specific name in computing: computational complexity. All functions have a computational complexity in time (i.e. they take time to execute) and in space. *(from Wikipedia)*

It follows that an ROC system may cache the results of all computation and, by suitable determination of equivalence, we may systemically eliminate redundant computation. In ROC this is called "systemic memoization".

It is beyond the scope of this document to discuss in depth, however it should be noted that the NetKernel platform employs a unique ability to automatically invert pass-by-value (push-state- transfer) to pass-by-reference with adapted scope (pull-state-transfer). This automatic inversion means that transient computational state acquires the same referential integrity as any other resource, therefore it is not possible to have out-of-band side-effects that would inhibit memoization.

The diagram below shows a small snapshot of the dynamic distribution of cached resources in a live system captured with the NetKernel Cache Heatmap Analytics Tool. Darker colours indicate stronger regions of caching. What is noteworthy is not that there are hotspots, rather that we always observe is that there are more coloured spaces than not. This indicates that the system is automatically discovering dynamically reusable state almost everywhere in our multi-dimensional architecture.

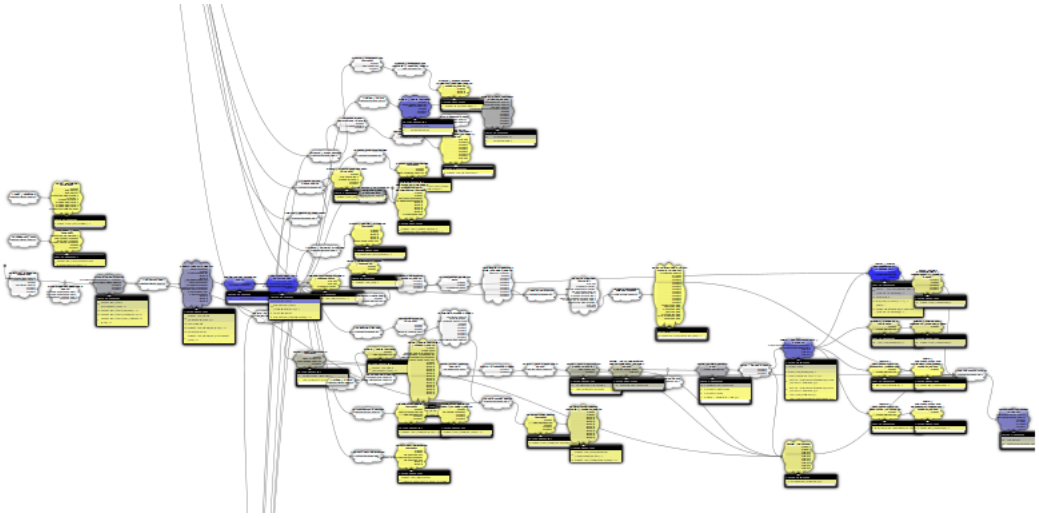

Figure 6: Cache Heatmap

The performance implications of systemic memoization are profound. Software systems are connected to the real world. The real world exhibits natural statistical distributions. ROC solutions are able to discover and manage the optimal real-time computational state

corresponding to the external statistical distribution demanded of it. In short, any redundant computation, at any scale can be tracked and eliminated.

It follows that, paradoxically to our experience, software that is indirectly coupled through the resource adaptive resolve-execute two-phase model of ROC is very often far higher performance.

We may understand this "paradox" with the following observation – the optimisation strategies of classical software (compiling, linking, JIT optimisation) have been concerned with making the transition between one function A and another B as efficient as possible – ROC simply says, in the real world, frequently the shortest route from A to B is to know that you are already at B.

For the academically inclined, it is worth noting that the effect of systemic memoization reaches beyond applied real-world problems. Our research indicates that the complexity of classically high-order algorithms can be dramatically modified when implemented in the ROC domain. The change may be achieved by implementing an algorithm in such a way that, rather than executing with local recursion (within the same code), we instead use extrinsic recursion (self-referential resource requests through the ROC domain). While the algorithm is identical, the route taken to determine the computational result is vastly different. Surprisingly, even algorithms as pathologically extreme as the Ackermann function, are more efficient in the ROC domain.

5.7 Linear Scaling with Mulit-core Architectures

Endpoints are stateless computation engines. Representations are immutable snapshots of the state of abstract resources. These properties allow the Web to implement load-balancing as a seamless adaptive scaling strategy.

ROC systems allow us to use the same strategy, both to scale out over distributed systems, but also, significantly, to scale-up on multi-core hardware architectures.

The NetKernel micro-kernel is able to load-balance execution threads efficiently over the available physical (or logical) compute cores. As more cores are added, the NetKernel platform automatically linearly scales up to utilize the increased capacity.

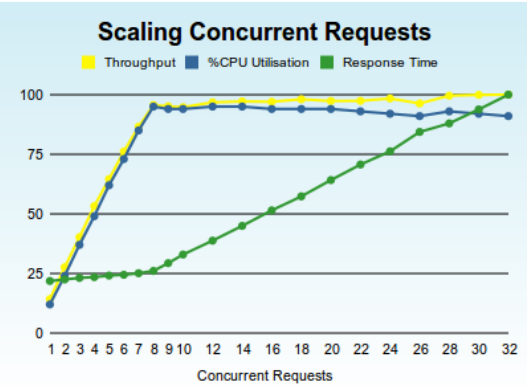

Figure 7: Scaling

5.7.1 Software Load Line

Figure 7 shows the results of the NetKernel platform's Linearity Analysis Tool. It can be seen that this example system has 8 cores. As concurrency is increased the system maintains a flat response time and linear increase in throughput. Once all 8 cores are occupied, the kernel fairly allocates the computations across the cores, throughput stays constant while the individual response time of a single request increases. There is no magic in parallel processing, the total computation is the sum of the available cores – when demand exceeds capacity you

must either increase cores or, as is straightforward in ROC, introduce engineering constraints to shape the computational distribution.

Figure 8 shows the Linearity Analysis of the same software configuration but this time deployed on a high-end virtual cloud compute platform operated by a well known cloud computing vendor. It can be seen that up to seven concurrent requests the system behaves linearly. However, when concurrency exceeds 8 requests things start to behave strangely – recall this is the same software stack as shown in *Figure 7*.

Surprisingly as concurrency increases further the total net throughput reaches equilibrium at about 50 percent of the maximum capability of the system. This strongly indicates that external throttling has been applied to this virtual machine (or put another way-we are getting half of what we paid for).

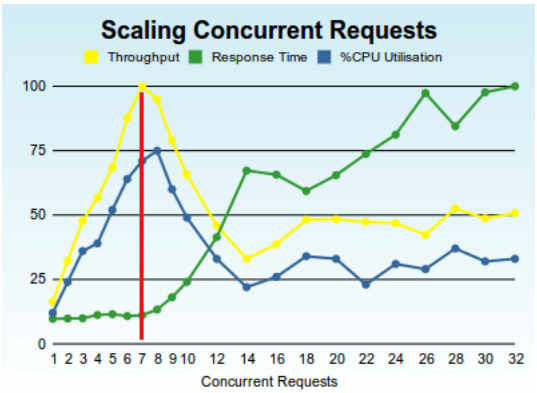

Figure 8: Linear Analytics of Physcial 8 Core System

It follows that a very sensible engineering strategy would be to configure the "software load-line" so that concurrency is locked at the peak throughput (7 concurrent requests). We shall see, following the discussion of the overlay pattern below, that it is trivial to introduce dynamically adaptive software throttles in NetKernel solutions.

The ability to introduce adaptive software throttles is an example of a large class of resource adaptive patterns that are readily available in the NetKernel ROC platform.

5.8 Set theoretic functions

5.8.1 Overlay

We must briefly introduce the overlay pattern since this construct is the basis for a large number of elegant engineering solutions.

Figure 9 shows the overlay pattern. It can be seen that it is an arrangement of two spaces whereby, somewhat like the import pattern, the routing of requests between the spaces is mediated by an endpoint (called the overlay).

An important characteristic of this pattern is that the space to which the overlay routes requests is entirely encapsulated by the overlay endpoint. In practice, the endpoint is even responsible for instantiating the inner space.

We can see that the overlay endpoint may be configured to project all of the resources of the inner space as resolvable entities to the outer space – somewhat as an import projects the resources of the imported space. This arrangement is called a "transparent overlay" - the requestor and inner space have no awareness that the overlay exists. However the overlay is there and can choose to arbitrarily deal with any requests that it resolves.

We can see that this pattern is very elegant for such tasks as providing secure trust boundaries in an architecture, for non-functional analytics capture, reporting, diagnostics etc

etc. It is also apparent that the dynamically adaptive software throttle that was introduced in the earlier section is in fact embodied as a simple transparent overlay.

Alternatively an overlay may choose to present its own "resource interface" to the outer space. That is, it may appear as one or more logical endpoints to requests issued in the outer space. If it chooses to accept a request, the overlay may be configured to actively transform the outer request to a different inner request. This arrangement is called an "opaque overlay".

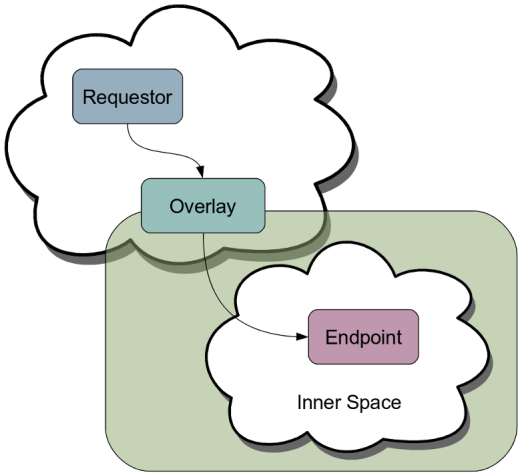

Figure 9: The Overlay Pattern

5.8.2 The Mapper

One extremely valuable class of opaque overlay is the "mapper". A mapper consists of logical endpoints each with a declared grammar. Upon resolving a request with a grammar, the mapper endpoint applies a transformation in order to map the identity of the outer resource to the identity of an inner resource. In practice both the grammar and the identity transform can be expressed with declarative expressions, resulting in a powerful construct that requires no formal programming language.

It is not at first clear, but the mapper is in fact forming a mapping relationship between one set of resources (outer) to another set of resources (inner). More formally the mapper pattern provides a formal set-theoretic function (mapping between sets) and, unlike the rigid imposition of functions determined by the syntax of formal programming languages, supports any of the well understood categories of mathematical function: injection, bijection, surjection.

When considering resource adaptive solutions, it should be noted that the implementation of the mapper provided with NetKernel has a very significant property - the configuration of the mapping is itself a resource. That is, just as for example with code-state transfer, the mapping is requested as a resource and therefore, if required, may be dynamically computed.

It follows that the mapper provides a very elegant means to formulate extremely powerful dynamically adaptive solutions. More particularly, when considering the formal definitions of stability and evolvability presented at the beginning, it can be seen that the mapper offers **I** to **R** and **I** to **I'** transformations and therefore, by our definition naturally facilitates long term stability while being a catalyst for continuous evolution.

5.9 Dynamic Import Patterns

It has been previously stated that within the ROC abstraction "everything is a resource". We shall now show how this is true even to the level of the architectural configuration of the software.

It is understood that a resource adaptive architecture must expected to be frequently (even dynamically) updated. For example one simple cais to deploy a new module with an enhanced capability to a system.

The usual way we would accommodate such a change would be to provide a reference to the new capability by importing it into a suitable architectural context – for example exposing an application to HTTP requests in a space containing an HTTP transport. The problem here is that we do not wish change in one part of our architecture to require change in another in order for it to be adopted.

In classical software, we have learned that the only way to introduce the capabilities of new functions is to declare an import in the location where we wish to access them. This is not the case in ROC.

The requirement for dynamically adapting configuration is common in ROC solutions.What is required is an import which is not statically declared, but which is dynamically configured, a "dynamic import". Everything is a resource, therefore the configuration of the dynamic import is also a resource.

On first consideration a dynamic import might appear to be rather similar to a regular import – for example in the case where only one import space is specified then they are identical. However recall that the configuration is a resource, therefore we are free to construct a solution in which the dynamic import configuration is itself dynamically computed.

Clearly there are many many ways this could be achieved – a very basic approach would be to have the configuration generated from an externally supplied state of a database, or another system, or even from the results of analytics of the system resources. The opportunity to provide dynamically adaptive solutions is very big.

One pattern that has proven to be extremely powerful is to apply a resource discovery and aggregation pattern. Imagine you have an architectural construct that is expected to offer a very long term stable structure – for example a transport and throttle arrangement. When we add a new capability to our system we want it to be exposed to the stable configuration – but we don't want to touch that configuration (either to preserve stability, or because it is none of our concern).

We can solve this problem by instantiating a dynamic import but we also provide a companion endpoint that dynamically computes the import configuration. The companion endpoint performs a space by space search making a request for a specified resource in every space in the system. If a space has the resource, and the state of the resource indicates that it is of interest, then that space's identity is added to an aggregation. When the search is complete the aggregated set of space identities constitutes the configuration for the dynamic import.

Figure 10 shows a schematic depiction of the Dynamic Import Pattern. One can see that all spaces that have resource X are imported.

Though it may appear straight forward to construct this pattern, from a classical software perspective it is impossible and cannot be done. Simply put, the dynamic import pattern is a non-deterministic computation in which the spaces and resource state of the software system are dictating the structure and architecture of the software system! This should not work, and indeed it could not without the properties of the ROC abstraction.

The dynamic import is a very powerful and efficient ROC construct. It is able to work because it is an intrinsically resource oriented pattern. When the search is performed, a space that is unstable (for example it is booting) may be deferred from the search, the

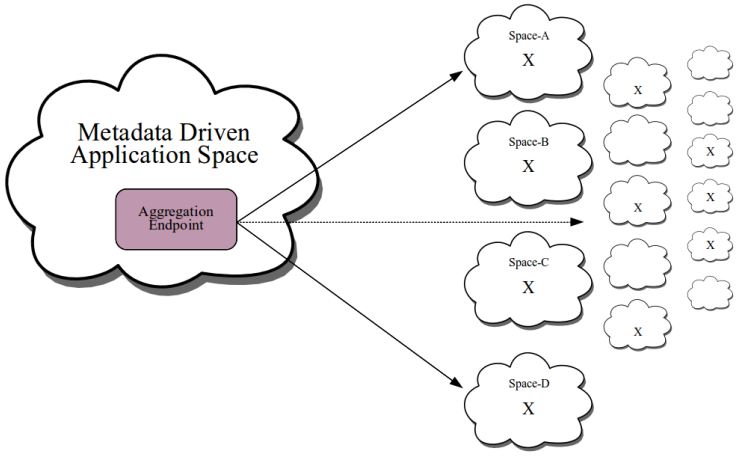

Figure 10: Dynamic Import Pattern

search proceeds non-deterministically, attempting to find import resources in spaces as and when they are sufficiently stable to be able to respond. It turns out that as the system evolves it anneals and the dynamic import resource is computable. Furthermore since the configuration is a resource and the space's import hook (the X in the diagram) are resources then we are able to compute a dependency model (as with the caching above). It follows that if the state of any of the X's changes – e.g. the space goes away, or someone decides not to be imported – then the aggregate import resource is invalidated and the import search aggregation is recomputed. The dynamic import is truly dynamic.

To the classical eye this may appear like witchcraft. However our experience and widespread use of this approach shows that it is a very elegant and robust engineering pattern. It raises profound questions about our expectations of, and the balance between deterministic and non-deterministic computation. Our experience indicates that given suitable boundary conditions a non-deterministic route to computational structure is entirely valid and desirable.

It will be apparent that this model of dynamic resource-oriented adaptive architecture is of great consequence in the implementation of resource adaptive systems.

5.10 Transreption: Entropy Minimization

In the discussion so far we have discussed the ROC abstraction but have not considered what form the representation of resource state should take. This is in contrast to the classical software model where the type of objects and member variables is often the first consideration.

In ROC the representation that is computed by an endpoint is not proscribed.

Frequently similar regions of a computational system will choose to adopt common structures for their representation. We call this a "resource model".

Similar economic interest in common exchange formats occur quite naturally in ROC solutions – in NetKernel there are for example collections of tools for processing XML,

images, RDF, strings, Wiki markup, HTML, database result sets, hierarchical structured data (HDS) etc.

This common typing alliance has a long heritage and can, for example, be observed in the interplay of tools in Unix that interoperate through the exchange of line-delimited ASCII files.

However within software there is much more opportunity for type variation than in a file-based operating system. How does an ROC solution avoid the long-term "type-death" of disparate endpoints being coded to different representational object models? This problem is especially pertinent to the long-term evolution of resource adaptive systems.

In the Web, the typing problem is less acute since the network transfer of state demands that ultimately all state must be serializable binary streams (just as in Unix all representations are binary file streams). Even so, the REST architectural style makes great play in the provision for user-agents and browsers to be able to perform content negotiation based on MIME-type preferences.

ROC also provides a form of content negotiation – however as with the ROC abstraction it generalizes beyond the recommendations of the REST style and is applicable to the internals of software where objects have very real and very varied structures.

In ROC an endpoint that constructs a resource request must specify its identifier, it may also provide an indication of a preference of representation type.

The resolved endpoint may choose to examine the preference and adapt its implementation in order to return the preferred type. However, it will be understood, that as a system evolves, the necessity to accommodate more and more (N) types leads to an N-squared increase in the complexity (and brittleness) of the solution.

It is far more desirable to permit a resolved endpoint to ignore the preferred type and to compute and return its representation in whatever form was deemed appropriate when it was created.

In ROC this potential problem is solved as follows. If the returned representation does not match the requested type, the NetKernel microkernel will step in. It will automatically construct and issue a new request, with a special verb, this request asks the spacial ROC domain to resolve an endpoint that will provide the necessary transformation between types.

However this mechanism cannot permit arbitrary transforms – recall that representation state must remain immutable in order to guarantee referential transparency (and unsynchronized concurrency).

Endpoints that perform transformation of representations must guarantee that the information of the resulting representation is identical (isomorphic) to the original. That is this is a lossless transrepresentation. This generalized requirement did not have a historical name and so in ROC the formal expression transrepresentation has, with use, become shortened to "transreption".

It follows that transreptor endpoints can be supplied such that as one resource model interacts with another the representations are dynamically transrepted.

At first it appears we have simply shifted the N-squared type problem to another place. However in practice transreptors may choose to accept a request even if they only match on the destination type, they may in turn request the incoming representation in a preferred form that they are able to process, in this case the kernel intervenes again and constructs another transreption request. It follows that the ROC abstraction facilitates automatic chaining of type transformation. In practice this property means that the N-squared problem is typically

simplified to linear complexity.

When all state is transferred to stateless endpoints it becomes apparent that many concepts in classical software become unified. For example, parsing a stream to an object model is a transreption, as too is compiling human readable code to machine executable byte-code, even serialization is a form of transrepresentation.

In fact transrepresentation is an essential requirement of any computational system – it's just historically we didn't have a uniform abstraction in which to place it. But we can go further.

If we recognize that ROC is providing a dynamic computational state space, we can understand that ROC's systemic memoization acts so as to minimize the total computational energy cost of a system. It follows that we may also understand that transrepresentation is acting so as to minimize the total information entropy in the system.

A transrepresentation transforms structure that is not suitable for an endpoint into a structure that is – that is, a high entropy structure is reordered into a low entropy structure.

Recalling our opening statement:

By the 2nd Law of Thermodynamics we know that entropy will always increase. The challenge facing long lived resource adaptive systems is to fight entropy with the least possible expenditure of energy (physical, computational, financial). We hope that it is apparent that the Resource Oriented Computing abstraction offers a unique ability to adaptively respond to change whilst requiring, as close as is practicably possible, the minimum energy expenditure.

6 Description of the NetKernel Platform

The NetKernel platform is a modular Java-based system. It requires only a standard Java runtime to execute and can operate with as little as 1-2MB of heap. *Figure 11* shows the schematic diagram of the NetKernel platform.

6.1 Microkernel

The core of the system is the microkernel – this implements and embodies the ROC abstraction. As we have learned above, the primitive elements are spaces, endpoints and requests. Just as HTTP provides the mediation of Web requests, so the NetKernel micro-kernel mediates ROC requests. Implementing the two-phase resolve-execute cycle inherent to ROC.

6.2 Caching

Working closely with the Kernel is the NetKernel cache. This cache provides systemic memoization. All computation at all scales are potentially cacheable in NetKernel.

A small bootloader is used to initiate the NetKernel system – typically a complete NetKernel application server configuration boots in 2 or 3 seconds.

Above the Kernel is a Layer0 – this provides specific technologies required to physically implement a working ROC abstraction. For example it provides support services to enable modularity, physical Java classloaders and several implementations of grammar that may be used by endpoint for request resolution.

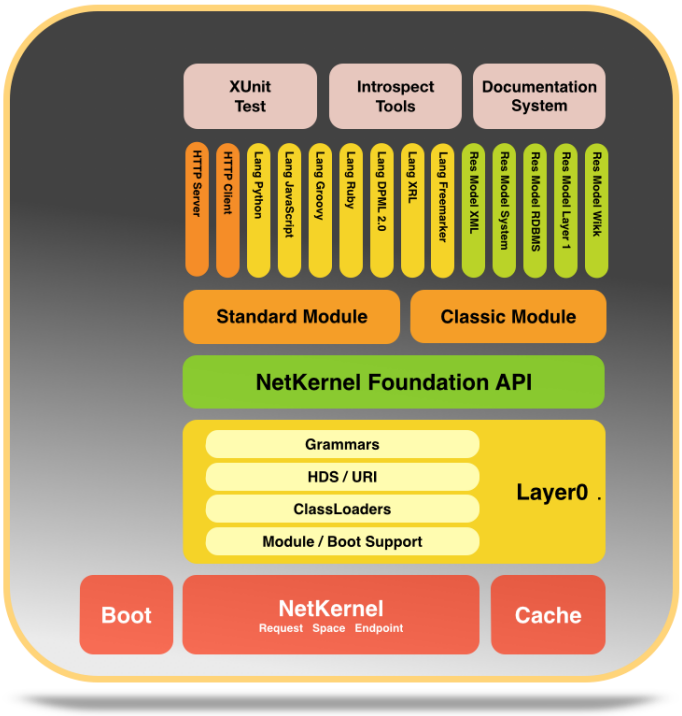

Figure 11: Schematic of the NetKernel platform

6.3 Foundation API

Above Layer0 is a the NetKernel Foundation API – this layer provides a Posix-like abstraction to ensure low-level details of the implementation are cleanly separated from the higher level space and endpoint implementations.

6.4 Modules

The Standard Module provides infrastructure to implement modules. A module is a physical container providing embodiments of spaces, endpoint and, as required, resources (such as code, configuration, static files etc etc). The Standard Module also provides specialized base-classes to aid the development of specific forms of endpoint. While specialism of endpoints is useful, it is important to emphasise that the core ROC system treats endpoints entirely generically and, as with the Web, the implementation of endpoints is a black-box to the Kernel and the abstraction.

The NetKernel system provides hundreds of modules. Modules provides services such as language runtime endpoints, client and server transport endpoints, domain specific resource models and resource transformation endpoints, domain specific language endpoints, vertical resource models etc etc. Modules may be hot deployed and dynamically reconfigured without any downtime of the core kernel. This property allows for evolvable dynamically adaptive systems to be deployed over very long time periods.

6.5 Self-supporting System

Finally, the NetKernel system is self-supporting. A wide set of system tools are implemented as resource oriented applications. For example tools are provided for testing, documentation, search, linearity analytics, cache analytics, system introspection and discovery, metadata aggregation and search, etc, etc.

A very powerful tool that can be used for development and for operational analytics is the Visualizer. The NetKernel visualizer is somewhat like a time-machine debugger in that it can continuously record the state of execution of the system for static replay and inspection. Unlike a typical debugger the Visualizer captures pre-existing execution state of the kernel and so introduces no performance overhead (Heisenberg effects) to a running system (other than the additional memory required to store the state). Furthermore, and unlike a debugger, the Visualizer also provides detailed measurement of the performance of the system – it is therefore also a time-machine-profiler.

All of the tools provided with NetKernel are resource oriented solutions – it follows that any capability offered by a tool is also a service that may be deployed in some other configuration. For example as part of a rich analytic solution or in reporting and management.

The NetKernel system is *evolvable*, and very *stable*. It has undergone continuous evolutionary change for over a period of more than ten years. Many tools and applications have required no adaption in that time.

6.6 Minimal Configuration

While the NetKernel platform is provided as a full featured application server, the core NetKernel system is designed to be embeddable as a micro-architecture in any Java container.

An embedded core system consists of approximately 1MB of jarred libraries and only requires an instantiation of a kernel (K) and a module manager. Any suitable configuration of modules can be initialised to form an ROC domain within the Java container. External code can construct and issue requests into the ROC system using the NetKernel Foundation API of a gateway endpoint.

The full range of ROC properties and patterns are available to the embedded ROC solution.

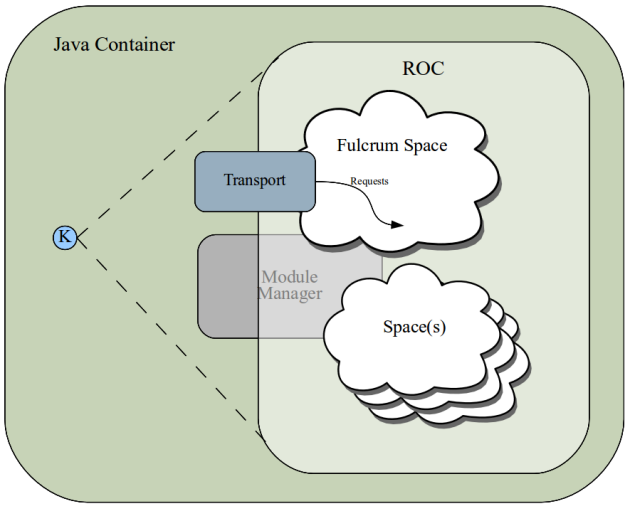

Figure 12: Minimal Configuration

Careful consideration of the core design ensures that all operating state is encapsulated, it is therefore possible to run multiple embedded instances in the same virtual machine.

Equally, since an endpoint is a software container, it is possible to run NetKernel within NetKernel offering the potential of virtualized ROC domains.

About

1060 Research Ltd. was spun-out from HP Labs in Bristol in 2002, pioneered a new abstraction in computing called Resource-Oriented Computing® in order to change the economics of software engineering. The team went on to develop NetKernel™ the most adaptable next generation platform. It has been deployed in some of the most demanding environments and in a number of industries including telecommunications, e-commerce, defense and education.

Download

To get NetKernel from GitHub requires a copy of Gradle and a Java JDK. Alternatively, to obtain a copy of the Enterprise Edition from the NetKernel Portal please register first here.

Glossary

Ackermann function In computability theory, the Ackermann function, named after Wilhelm Ackermann, is one of the simplest and earliest-discovered examples of a total computable function that is not primitive recursive. All primitive recursive functions are total and computable, but the Ackermann function illustrates that not all total computable functions are primitive recursive. – The Ackermann function, due to its definition in terms of extremely deep recursion, can be used as a benchmark of a compiler's ability to optimize recursion. The first use of Ackermann's function in this way was by Yngve Sundblad.[4]. 15

memoization an optimization technique used primarily to speed up computer programs. 14, 15, 21

REST **RE**presentational **S**tate **T**ransfer. "The REST style is an abstraction of the architectural elements within a distributed hypermedia system. REST ignores the details of component implementation and protocol syntax in order to focus on the roles of components, the constraints upon their interaction with other components, and their interpretation of significant data elements. It encompasses the fundamental constraints upon components, connectors, and data that define the basis of the Web architecture, and thus the essence of its behavior as a network-based application".[1]. 3–5, 11

transreption trans-representation, transformation of representations. Information of resulting representation is identical (isomorphic) to original. Lossless transrepresentation.. 20

References

[1] Roy T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. dissertation, 2000.

[2] Yiannis N. Moschovakis. *What is an Algorithm?, in: Mathematics Unlimited*. Springer, 2001.

[3] Arnold Rosenberg. *The Pillars of Computation Theory: State, Encoding, Nondeterminism*. Springer, 2010.

[4] Yngve Sundblad. *The Ackermann function. A Theoretical, computational and formula manipulative study*. BIT Numerical Mathematics, 1971.

[5] Jan van Leeuwen and Jiri Wiedermann. *The Turing Machine Paradigm in Contemporary Computing, in: Mathematics Unlimited*. Springer, 2001.