The Compositional Architecture of Distributed Systems

Robert J. DuWors

Connected Systems Group

January 22nd, 2006

A New Way of Looking at Distributed Systems

Foreword

Compositional Architecture is a unifying theory. We currently suffer from the onslaught of divergent architectures: Object Oriented Architecture, Component Oriented Architecture, Data Oriented Architecture, AJAX Oriented Architecture, and Service Oriented Architecture. With so many "Orientations", no wonder that we rapidly become disoriented! None of these can explain the way the web works today, every day. Compositional Architecture looks for what underlies all of these to fit them into a bigger picture, like the pieces of a jigsaw puzzle.

Compositional Architecture looks beyond them too. Compositional Architecture can explain what goes on by loading a simple web page. Compositional Architecture can unite AJAX clients with the requisite highly personalized security model in a distributed environment. Compositional Architecture thus provides distributed system patterns to structure the backend (network) servers accordingly. This unification allows AJAX clients to seen clearly as User Agents that provide yet another kind of network service. Compositional Architect also deals well with the manipulation of Business Processes and Service Orchestration.

And that's just the opening act for Compositional Architecture as found in this Technical Report.

Contents

Foreword i
The Compositional Architecture of Distributed Systems 1
Abstract 1
Overview of Compositional Architecture 1
A Brief Introduction to Compositional Architecture 8
The Six Basic Building Blocks of Compositional Architecture 8
The Four Elemental Units of Compositional Architecture. 8
Network Nodes and Links 10
Instances, Processes and Meta-levels 11
The Acquisition and Automation of Knowledge 14
The Six Fundamental Relationships 15
Network Topology 16
Protocols 16
Binding: Transclusion and Metabinding 17
Open World Assumption 18
Temporal Semantics 20
Security 21
The Six Degree of Correctness 24
The Scope of Compositional Architecture 24
The Limits: A Little Humility about Living on the Fault Line 25
Appendix A: Thoughts on Service Oriented Architecture as a Major Piece of the Puzzle 29
Messages and Behaviors 29
The Two Major Architectures of Protocols: Data Centric and Behaviorally Centric 30
The State of Protocols 31
Who's Winning, Who's Losing and The Importance of Picking the Right Battle 32
Building and Re-using Application Protocols 33
Appendix B: Consideration of Object Use and Abuse in the Architecture of Distributed Systems ... 36
In the Beginning and Thereafter 36
Drowning the Baby in the Bath Water 38
The Object of Persistence 38
The Fine Art of Dynamic OO Composition 39

Figures

Figure 1: Sources of a Web Page 3
Figure 2: Composition of Business Rules - A Wireless Invoice Rating Example 4
Figure 3: The Dynamic Composition of a Business Process 5
Figure 4: The Basic Building Blocks of Compositional Architecture 9
Figure 5: Examples of Common Architectural Meta-Models 13
Figure 6: Topology Changes with the Level of Abstraction 16
Figure 7: Comparison of Object Oriented and Compositional Architectures 17
Figure 8: The Functional Architecture of an Application Service Oriented User Agent (SO-UA) 18
Figure 9: The Evolution of User Agent Architecture showing "Scientific" and "Common" Names 19
Figure 10: An Example of Security Policy as the Composition of Rule Sets 22
Figure 11: The Traditional Meaning of Roles, Groups and the Access Control Matrix 23

Note: The Appendices can standalone from the main report. In particular, *Appendix A: Thoughts on Service Oriented Architecture as a Major Piece of the Puzzle* extends the Protocols Section. Even a cursory review of this paper should include *Appendix A,* which may also be used as an introduction to the main body.

10/5/2007© 2006, RJD

The Compositional Architecture of Distributed Systems

Robert J. DuWors
Connected Systems Group
January 22nd, 2006

"The purpose of Architecture is the answering of Application Domain Wants with IT Solutions." [1]

"The function of IT, pure and simple, is the automation of knowledge." [2]

Abstract

This paper introduces the foundations of Compositional Architecture targeted at the understanding and design of distributed systems. Compositional Architecture provides a framework for automating distributed knowledge. This paper treats composition as the key operation to support this goal and introduces the notion of Metabinding coupled with Transclusion. Three major themes as decribed in this report define Compositional Architecture: The Six Fundamental Elements from which all resources are constructed, The Six Fundamental Relationship that apply to all distributed systems, and The Six Degrees of Correctness. Consideration is given to Compositional Architecture as a general model of computing with "fractal-like" self-describing similarity at all levels of abstraction. Compositional Architecture is seen to subsume and surpass Object Oriented (OOA), Component Oriented (COA), and Service Oriented Architectures (SOA). Finally, Compositional Architecture signals a significant shift in established practices: Compositional Architecture is a way of thinking.

Overview of Compositional Architecture

Compositional Architecture is knowledge oriented. Compositional Architecture views software as a media for the acquisition, storage, modification, and active use of knowledge. By their very nature, distributed systems place knowledge content throughout interconnected networks. Naturally, the central issue in distributed systems architecture is the use of this composite knowledge. Functionality results when the right computations come together with the right persistence (memory) at the right time in the right places resulting in the right outcomes reaching the right destinations[3]. Composition is the essence of this gathering together and spreading out of distributed knowledge content.

Compositional Architecture describes distributed systems in which the basic entities are dynamically composed and recomposed out of elemental network resources as needed on the fly. All entities within the network communicate with each other by means of appropriately defined protocols for the retrieval, transfer, loading, and distribution of network resources. The meaningful composition of these far flung network resources expresses the dynamic knowledge content to be used as needed in order to produce the appropriate results

Dynamic resources in Compositional Architecture are composed of four elemental types[4]:

Units of Computation	(uC)	e.g. programs, processes, scripts, rules, DNA[5]
Units of Units of Persistence	(uP)	e.g. data, memory, DLLs, archives
Units of Distribution	(uD)	e.g. messages, parameters, mRNA
Units of Transduction	(uT)	e.g. I/O, sensors, User Interfaces

10/5/2007© 2006, RJD

The distributed system's automated knowledge content is represented by these elemental units singly or jointly. New knowledge arises from computations which generate, transform and compose the fundamental units together into new entities, making for an extremely fluid notion of form and function. To be effective, these units must fully expose all of the knowledge content within them. Any form of "information hiding" is strictly anathema. All knowledge content must be visible so that it may be recomposed in whole or part as needed. The only exception to universal visibility is any knowledge content explicitly hidden at that moment by a security policy[6]. The secured content thus literally drops out of sight and intentionally becomes unusable within that security context.

Since security is yet another piece of automated knowledge, Compositional Architectures are capable of incorporating security policies and rules as parts of themselves. This reflects the ability of Compositional Architectures to be self defining. Security policies are made out of the same four elemental units and subject to the same highly dynamic composition[7]

The basic rules of Compositional Architecture govern how the four elemental types can be assigned (housed) in named locations (nodes) and transmitted across addressable channels (links) in a network to be combined and transmuted. Because units of Computation can flow as easily as units of Persistence, Compositional Architecture uniquely supports "rule flow" as easily as "data flow."[8] Rule flow enables building systems that provide both high performance and highly flexibility – which is otherwise oxymoronic in today's relatively brittle Object, Component and "Service" Oriented Architectures.

Compositional Architecture makes extensive use of three types of binding. The first two, early and late binding, are traditional, whereas the third, meta-binding, distinctively belongs to Compositional Architecture. Early binding pre-defines linkages well before use e.g. binding relational database schemata to instance data. Late binding establishes linkages at the time of use, e.g. program interpreters, object instantiation, polymorphism, and class loader binding. Meta-binding is the third type and it is new. In meta-binding, the entities and the linkages between the entities come into existence through composition of the four elemental types at the time of use according to the meta-rules outstanding at that moment, which themselves may be subject to dynamic change.

The fetching of remote content for local inclusion is known as "transclusion" in web terminology. Transclusion is one of the central concepts in Compositional Architecture[9]. Remotely sourced transclusion plus local content provide the raw material for composition. Leading to the slogan:

"Don't just reach out and touch --- reach out and make it a part of yourself!"

Transclusion is another method of composition in addition to modularity. Modules combine only through their interfaces. This describes software modularization and componentization. It appears superficially to completely describe hardware: the ICs on a board communicate only through their interfaces. While this is true of the final product as a hardware process, it is manifestly untrue of the design process that created the rules for making the hardware components in the first place. Hardware designers use large collection of "cells" that they cut and paste into the final design. In fact, "cut and paste" nicely describes transclusion.

Anyone who has ever used a macro language, a template, a text editor, a mailing list, a Power Point presentation, or a word processor already possesses intimate experience of tranclusion. At the programming level, rather than that of the final software, the whole effort would collapse without "cut and paste[10]".

10/5/2007© 2006, RJD

In the case of Internet technologies, Uniform Resource Indicators (URIs) lubricate the ability to reach out to network resources by means of the appropriate references and protocols. URIs allow resources to be retrieved and redistributed as needed. For example, the following is a partial list of HTML tag types that use URIs to dynamically compose an ordinary web page:

Anchor	Embed	Form	Input	Link
Meta	Body	Frame	iframe	frameset
script	Object	Head	applet	Doctype

Even a simple web page is, of course, made out of multiple units of Persistence and units of Computation delivered on demand from multiple locations across the Internet by units of Distribution and shown to the user by units of Transduction rendered via the browser's User Interface. Thus, Compositional Architecture in comparison to OO, Component, and pure Hypertext models, offers a much better architectural perspective to explain what we actually do and what actually happens every day on the Internet. Compositional Architecture subsumes these other architectural models as special cases and thus surpasses rather than eliminates them. Similarly Compositional architecture subsumes Service Oriented Architecture (SOA) under the more generalized banner of Protocol Centric Architecture (PCA). But is PCA is not just a special case, Protocol Centric Architecture is an integral part of Compositional Architecture, forming its beating heart.[11]

User Agents are the peers of network computing services. User Agents represent the user to the network and the network to the user. The network services may concentrate the resulting processing in one place or spread it out widely, and decide if traces of it should be left behind. One User Agent may use the same service from any number of interchangeable back end servers. Conversely, a server may go looking for any number of User Agents to represent its services to users, i.e. the User Agent IS a network service, one the mediates between the end user and the rest of the network services. While "clients" in client-server architeture are inevitably 1:1 with servers, User Agents are n:m with services. AJAX allows User Agents to be delivered on demand when needed.

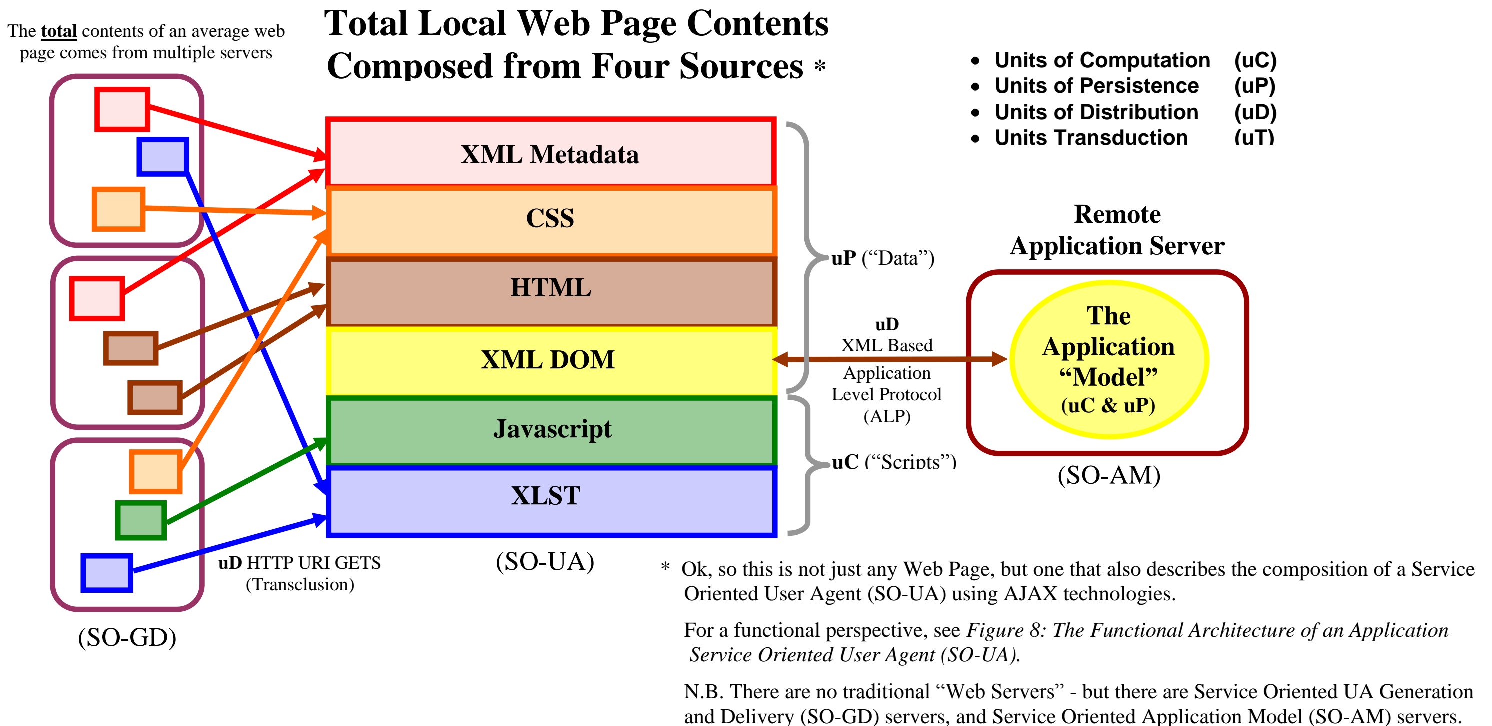


Figure 1: Sources of a Web Page

10/5/2007© 2006, RJD

URIs are to Internet technologies are as symbol tables are to compilers and loaders, i.e. the primary linkage mechanism that enables dynamic composition. Thus, both types of linkage mechanism fall within the scope of the Compositional Architecture framework as do Application Program Interfaces (APIs), and Application Level Protocols (ALPs). CSS, Javascript, VBScript, XLST, Java and assorted commercial plugins such as Flash make further use of URIs to compose, display, pass parametric data, and interact with the current web page[12]. Special mention should also go to the upcoming importance of XPath and the OASIS eXtensible Resource Identifier (XRI) proposal.

Of course HTTPXMLRequest has single handily stirred up the Asysnchronous Javascript + XML (AJAX) hornet's nest where a dynamically loaded user agent resides in the bowser on demand. HTTPXMLRequest communicates on behalf of the user agent's internal elements that speak the network Application Level Protocol (APL). HTTPXMLRequest has no direct role in the other major aspect of the User Agent that deals with the localized display and user interaction, i.e. the User Interface (UI) functionality. This clear separation of concerns, network application semantics from final rendering and user interaction, has sent shockwaves through the web development community.[13]

At a higher level of Application Architecture, the following diagram shows the use of Compositional Architecture in the rating component of a high performance telecommunication billing system:

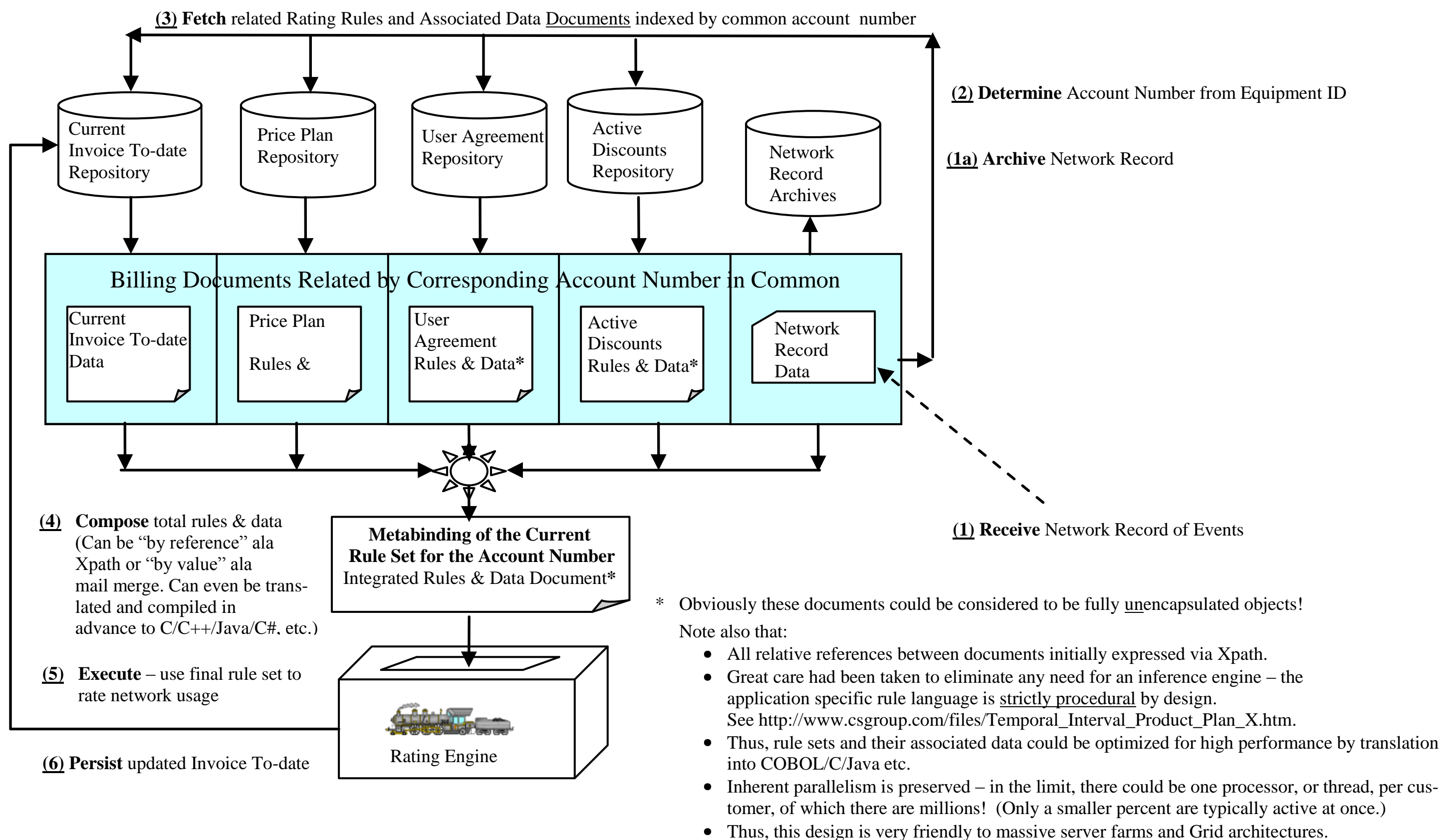


Figure 2: Composition of Business Rules - A Wireless Invoice Rating Example

Because so much dynamic composition is going on, Compositional Architecture often makes extensive use of "meta" relationships such as "describes" or "defines" between layers in the application/system stack as exemplified by the 4 level UML meta-meta layers. It also employs the more traditional "uses" relationship typical of statically layered architectures such as the infamous 7 level OSI model of data communications, and object style interaction.[14]

10/5/2007© 2006, RJD

Employing Compositional Architecture to compose and execute dynamic Business Processes is shown next.

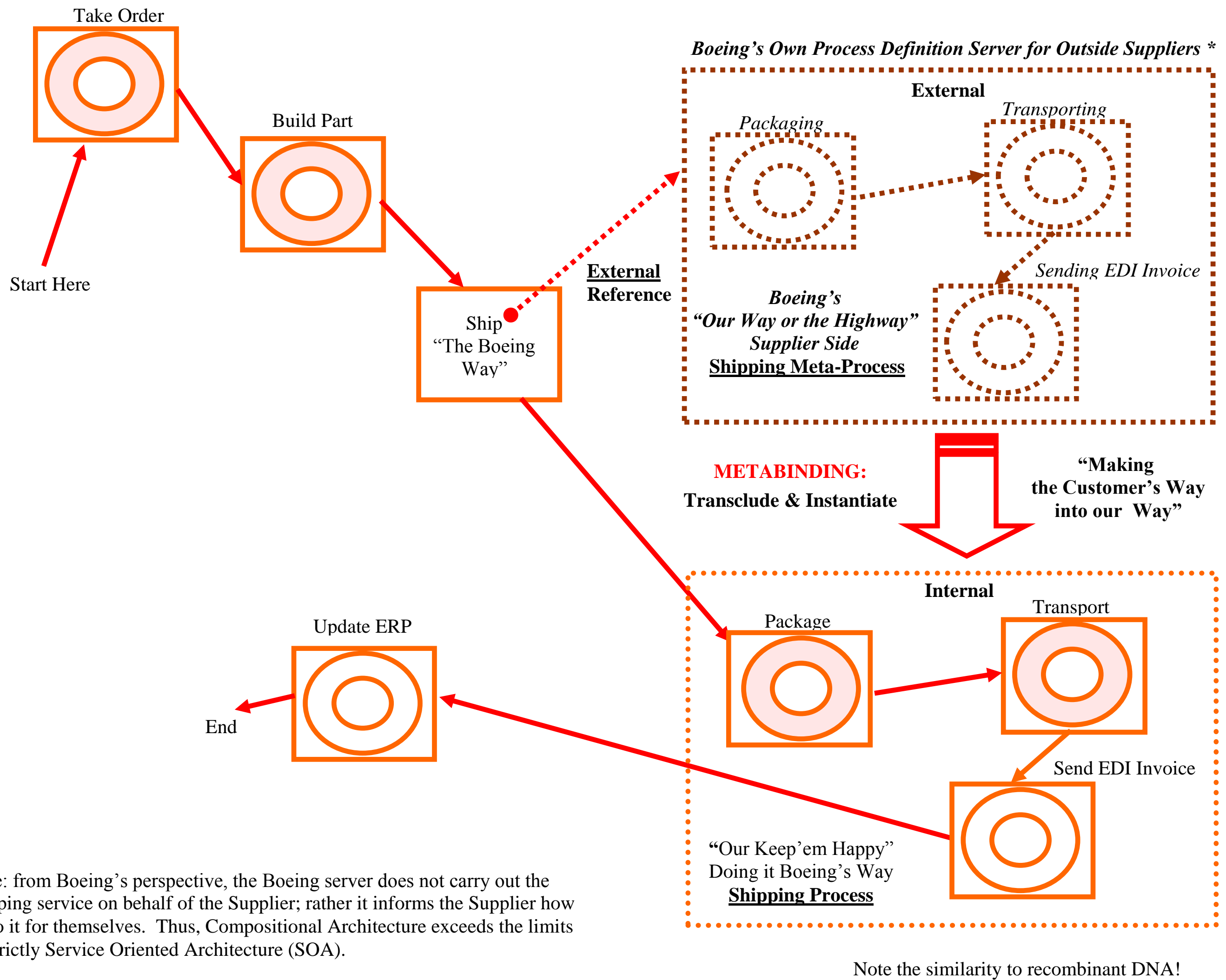


Figure 3: The Dynamic Composition of a Business Process

Much of modern Object Oriented methodology focuses on "patterns". The Gang Of Four (GOF) kicked off this movement with respect to Object Oriented software by defining a collection of low level object patterns. Roger Sessions has used Component Architecture to define a "Software Fortress" model suitable for enterprise class applications.

But - Object and Component models can not be self defining by their fundamental nature, and required the human designer to go outside of them to use other ways of interpreting the application of the pattern. Hence these architectures must generate mountains "design artefacts" and extraneous "architectural models" which ultimately get discarded. However, Compositional Architecture can define these patterns from within, and the linkage from them to the pieces composed and controlled by the pattern's rules of interaction[15]. This is true even if the pattern's rules sets are widely distributed[16]. See *Figure 7: Comparison of Object Oriented and Compositional Architectures.*

10/5/2007© 2006, RJD

Uniquely, Compositional Architecture allows for self defining and hence self constructing systems. The very same higher meta-level(s) can serve multiple roles as a design definition, implementation scaffolding, a deployment descriptor, and a run time control. There can be cascading levels where a more generic architectural template creates a specific architectural instance which then brings about the next level of elements in the architecture.

Certainly multi-level definitions and instantiations of this type make a great deal of sense and offer a great deal of power. Because these higher level descriptions can be "constructive" (executable) in the formal sense of the word, they are not wasted. The more this approach permeates downward[17] in system detail, the more flexible the resulting systems. Of even greater merit, this promotes higher levels of knowledge representation throughout the entire system. And thus moves in the direction recommended by Albert Einstein:

"We can not solve our problems with the same level of thinking that created them."

---

[1] Yes, Wants. We don't really need more cars, yet another cell phone, deadlier weapons of mass destruction, or huge shareholder dividends – we simply choose them or not. We can, and in some cases should, do without. So, we don't really have any absolutely necessary "requirements", we only have wants strong enough to act on. Rather than sorting "needs" from "wants", we really need to sort "wants" from "wishes" – only the wants are strong enough to be self justifying calls to action. "Requirements errors" are quite real: a system that meets some arbitrary notions of "needs" and "requirements" but not wants, is worse than doomed – it will generate serious negative repercussions.

[2] This view aligns with the pioneering articles of Phillip Armour. See *The Acquisition and Automation of Knowledge* in the next section. By "knowledge" we mean awareness of the universe, how to manipulate things in the universe, or the same for abstract systems. We will not quibble with epistemological niceties which draw distinctions between "knowledge being only in the mind of an intelligent entity" versus "mere knowledge representation contained in an automated system." In all humility, sooner or later machines may be better at knowledge than we are.

[3] A.k.a. "The Six Degrees of Correctness."

[4] Summarized in *Figure 4: The Basic Building Blocks of Compositional Architecture*.

[5] As Philip Armour noted, DNA was the first major form of knowledge representation, that of how to build a living organism. DNA is a type of meta-process definition, which describes the production of proteins. DNA demonstrates the composing of computational elements by recombining DNA extracted from two sources (parents) to produce meta-processes containing new life definitions (children). Business processes composed from multiple sources have similar aspects. mRNA shows messages from the DNA archive to build proteins. DNA demonstrates the power of genetic algorithms!

[6] All knowledge content in lieu of any security policy is visible. The security policy may choose to enforce "default permit" (worse) or "default deny" (better). Compositional Architecture in itself does not impose any impediments to the security policy. The rules of the security policy(s) are dynamically composed and applied as needed. The OASIS eXtensible Access Control Markup Language (XACML) employs the dynamic rule set(s) approach, as do most databases which support "views" and permissions. In essence, a computational paradigm should not impose any ot its arbitary restrictions on the security mechanism. Required knowledge content must be made visible as appropriate, i.e. available for transmission (uD), computation (uC) , persistence (uP) and interaction with the outside world (uT). Selective is good, but coyly hidden is not. Thus, Compositional Architecture is very "security friendly." In contrast, Object Orientated and Component architectures tie the hands of the security policy with their own unavoidable "information hiding" that may allow indirect and undetectable "information leaking" contrary to the security semantics of the application.

10/5/2007© 2006, RJD

[7] As shown in *Figure 10: An Example of Security Policy as the Composition of Rule* Sets. In particular, the security rules applicable to an individual are simply a subset of the total rules of "personalization", namely those personalization rules that restrict functionality and visibility. This observation forms basis of the Distributed Capability Model and its specialized form, the Bifurcated Distributed Capability Model suitable for use with dynamically loading User Agents such as AJAX clients. See *Figure 11: The Traditional Meaning of Roles, Groups and the Access Control Matrix* and the preliminary The Distributed Capability Security Model at www.csgroup.com. The XACML standard from OASIS provides a sophisticated model by which to apply the principles of dynamic composition of security rules, although XACML still seems to still be Access Control List (ACL) oriented.

[8] As an example, see Applying Intervals to Actions in a Document Rendezvous Model to Support Billable Event Rating: Beginnings of a Compositional Architecture on the Writings page at www.csgroup.com.

[9] It is reasonable to view a protocols as a series of transclusions, see Protocols in the next section.

[10] Perhaps "copy and paste" more exactly matches transclusion, but "cut and paste" has more cache!

[11] It's debatable whether or not Compositional Architecture subsumes Process Oriented Architecture (POA) in the sense of the Pi Calculus. Despite apparent differences, the fundamental units of uC, uP and uD can map onto "processes." It thus appears that Compositional Architecture can incorporate POA as one of the available formalisms. Protocol definitions, static and dynamic, are one area where POA might excel for rapidly changing Business Processes. Compositional Architecture springs forth from these fertile grounds. It appears that Microsoft has gone in this direction with Windows Workflow Foundation (WWF). WWF does not yet seem to have a tie into SO-UA AJAX clients and their SO-AM and SO-GD back-ends, as that requires a more general framework like Compositional Architecture.

[12] It is well worth noting how multi-linguistic a single web page can be. Furthermore, the computing formalism behind the languages varies radically: most are imperative, but XLST is truly functional and HTML is purely descriptive, some are object oriented and some are not. Both the client and server sides of internet technologies employ an unprecedented bevy of languages at one time. The notion of "knowledge content" is unavoidable in order to bridge them.

[13] This reality initially prompted the effort to codify the architectural principles that have evolved into Compositional Architecture. It was obvious that something new was going on outside of the usual OO and Component Architectures, and Service Oriented Architecture was not complete enough to describe it. However, at that time neither the author nor the developer who had independently discovered AJAX techniques and is also an accomplished architect, could say what. Thus began this journey that has led to many more destinations than originally imagined.

[14] See Figure 5: Examples of Common Architectural Meta-Models.

[15] Databases very much fall within the scope of Compositional Architecture, making good exemplars. In theory and practice, Relational Database Management Systems (RDBMS) do a lot more than just store data. They also store a surprising amount of computational ability in the form of triggers and stored procedures. A RDBMs without these would be greatly reduced in value to the point of being competitively crippled. SQL is a very active language for manipulating both meta-level schema and instance data. Without that ability, Databases would face the same problem as first order logic – simple predicates systems can not create themselves and hence require an external means to come into existence. To get off the ground logically, you need "second order" meta-logic, Goedel not withstanding. Meta-levels are powerful, even if they sometimes have surprising consequences.

[16] All too often today's generation of "business rule engines" and "business process orchestration" force the rules to live in only one place at a time. But so not in XACML. XACML gets rule distribution right, or at least some predefined and open ended versions of it.

[17] Or, upwards for that matter. The more it permeates throughout the design and implementation, the better. The literal building of systems by meta-levels in many ways is the ultimate expression of "literate programming" – software that both people and machines can read.

10/5/2007© 2006, RJD

A Brief Introduction to Compositional Architecture

In pursuit of the automation of knowledge by IT systems to solve application wants, Compositional Architecture shifts attention from the fixed structure of functionality towards the active composition of functionality. More abstractly, content takes precedence over topology.

In its basics, Compositional Architecture features the following:

- The four elemental types provide the building blocks to compose the knowledge content.
- The nodes and the links of the network provide the means to house and to transmit them.
- The Six Fundamental Relationships characterize the collection and re-distribution knowledge within a distributed system.
- The “Six Degree of Correctness” describes correct operation.

The Six Basic Building Blocks of Compositional Architecture

The fundamental building block of Compositional Architecture are the Four Elemental Units of Compositional Architecture plus network nodes and links.

The Four Elemental Units of Compositional Architecture.

Compositional Architecture recognizes distributed systems as being made from four fundamental types:

1.	Units of Computation	(uC)	e.g. programs, processes, scripts, rules, DNA
2.	Units of Units of Persistence	(uP)	e.g. data, memory, DLLs, archives
3.	Units of Distribution	(uD)	e.g. messages, parameters, RNA
4.	Units of Transduction	(uT)	e.g. I/O, sensors, actuators, User Interface

These four basic units are fundamentally not reducible to each other at any one time[1]. But they all can be combinatorially joined to each other (composed) and they can be transmuted by units of Computation.

10/5/2007© 2006, RJD

Figure 4: The Basic Building Blocks of Compositional Architecture

Fundamental Units	Abbreviation	Description	Action Type	Interaction
Units of Computation "programs, scripts & rules"	uC	Can change their own state (compute) and the state of all fundamental units. Can only be active when resident at a node. Can send uD outward from node, and receive uD inward to node, across communications channels (links). Can send uT outward from edge nodes (export) or receive uT inward (import) at edge nodes.	Active	• Can Create, Read, Update, and Destroy (CRUD) all fundamental units: uC, uP, uD, uT • Can be persisted by uP • Can be carried by uD • Can put in and take out all fundamental units from uD • Can import and export uT
Units of Persistence "data and memory"	uP	Preserves unaltered state of all fundamental units until externally transformed to new state or destroyed by uC.	Passive	• Can make persistent all units: uC, uP, uD, uT • CRUD is by uC • Can be carried between nodes by uD
Units of Distribution "messages"	uD	Carrier mechanism between nodes: information in the form of the fundamental units that can be sent or received from node to node. Can only be carried across links between nodes. Can only be filled with fundamental units and transmitted, or received and emptied by uC at each node.	Passive, except to cross links	• Can carry all fundamental units: uC, uP, uD, uT • CRUD by uC • Filled with fundamental units and emptied by uC at each node
Units of Transduction "I/O including UI"	uT	Define the "inside" from the "outside of the network: represents information in the form of all fundamental types destined to or arriving from outside of the network at edge nodes. Can only sent or received by uC at edge node. Can only be created by uC and consumed externally (export) or created externally and received by uC (import). Carried inward and outward by some from of channel that is NOT internal to the network, i.e. not by a network link.	Passive, except to import or export externally to the network	• Can contain all fundamental units: uC, uP, uD, uT • Can be created by entities external to the network (import) • Can be created uC (export) • Can be received (import) or sent (export) from edge node by uC • Can be read and updated by uC • Can be persisted by uP • Can be carried by uD • Can be consumed by entities external to the network (export) • Can be destroyed uC (import)
Nodes "aka host"	uN or "node"	Site of all work done: all active instances of uC & uP live in nodes. Connected by links in network topology. Communicate with other nodes via protocols.	Active	• House all active instances of uC, uP, send/receive uD, and uT. • Talk to other nodes across links using uD, and outside world with uT if edge node/transducer.

10/5/2007© 2006, RJD

Compositional Architecture employs a very robustly "open world" model that concedes no distributed system is "an island (of automation)" unto itself, nor can any architecture by itself be fully complete in isolation. Most fundamentally, all knowledge expressible and hence capable of automation by a distributed system must be represented in terms of the four fundamental units of Computation, Persistence, Distribution, and Transduction, plus the network nodes and links that taken together form the basic building blocks of Compositional Architecture.

In many ways, Compositional Architecture has a "fractal-like" property of being self-similar at all levels of granularity. The same six fundamental entities always apply, even though the items fulfilling the roles may change such servers, databases, routers, objects, components, spread sheets[2], rule engines, dynamically composed and transformed XML entities, and right on down to the level of combinatorial logic circuits.

Key to advancing the automation of knowledge is the ability at multiple meta-levels to manipulate and to compose distributed units of Computation and units of Persistence. What exists as data that is open to manipulation at one level can be metadata to a lower level, e.g. an editable form letter template is meta-data to a completed form letter[3]. Another good example is the server-side generation of browser display content, which may or may not be such a good idea depending upon how the meta-relationships are architected.[4]

Network Nodes and Links

Networks are made out of two basic units:

1. Network Nodes (Node) e.g. computers, modules, rule engines
2. Network Links (Link/Channel) e.g. busses, data links, calling stacks

Following well-established convention, a network is composed of addressable places called nodes, and addressable edges that represent communications channel linkage ("connectivity") between the nodes. Nodes may be such things as a computer, a disk drive, a router, a database, a process, or a dynamically composed entity that exists briefly for one transactional stage and then evaporates leaving only traces of itself behind much like a short-lived sub-atomic particle. Thus, nodes and links may be dynamically created and destroyed.

Nodes are containers whose total content is composed dynamically out of all four fundamental units by treating composition as a function of time. Units of Computation and units of Persistence with units of Exchange and units of Transduction are composed into the appropriate permutations:

$$\text{node total state}_T := \sum_i uC^i_T \;\&\; \sum_j uP^j_T \;\&\; \sum_k uD^k_T \;\&\; \sum_p uT^p_T$$

Compositional Architecture strongly emphasizes an "open world" ontology[5]. An edge node has linkage to the outside world whose exact nature may lay beyond the scope of distributed system design. Units of Transduction flow between the edge node and the outside world. Thus all edge nodes are transducers, typically between the digital world of the distributed system and something outside that is often "analog" (continuous) in nature: music, video, sensors and actuators, and especially the human perceptual system. The edge node literally represents the boundary of the distributed system with the rest of the universe.

This use of edge node differs from traditional network architecture where an edge node is defined topologically in terms of those nodes that route to no other nodes. However, in Compositional Architecture, a router at the topological center of the Internet that has a human interface would nonetheless be an edge node, especially considering the security issues at stake. A "headless server" would not be an edge node, but an automated telemetry receiver station would be.

Of particular note, Compositional Architecture recognizes user agents to be a very important type of edge node that has the critical functionality of representing the user to the network and the network to the user. In the limit, given a network of a single node, the user agent would still be an edge node because of its external contact to the outside world!

A channel is anything that can carry information between nodes, such as a data communications circuit, in memory message passing, or a subroutine call stack a.k.a. Application Program Interface (API). Channels and nodes can be dynamically created and destroyed as desired, thus network topology is also dynamic.

Only units of Distribution (messages) can transverse the links. Messages can carry much more than just "data" but also "program and rules" and "I/O" and even other "messages", e.g. uD(uP), uD(uC), uD(uT), even uD(uD). Units of Transduction can carry whatever is meaningful to the outside world and receive in whatever is useful from the outside world – provided the uT can be interpreted on the network side in terms of the basic units of Persistence and units of Computation.

Similarly, units of Persistence can contain any of the other units, e.g. uP(uC), uP(uD), uP(uT) and even uP(uP).

Only units of Computation can create, retrieve, update, and destroy (CRUD) all the other fundamental units and themselves. Only units of Computation can "package" or "transform" the contents of the other units and themselves, e.g. $uC_1(uP) = uC_2$. And only units of Computation can send and receive units of Distribution and units of Transduction. Basically, only units of Computation are active, or can be active. All the others are passive, with the exception that units of Distribution also have the ability to traverse links and units of Transduction can enter from and leave to the outside world.

Instances, Processes and Meta-levels

The interaction of units of Computation and units of Persistent in the sense that Robin Milner used in his 1991 Turing acceptance lecture give rise to interesting consequences when the unit of Persistence (aka "memory" or "data") outlives the immediate computation hereby preserving "state" between computational instances[6]. At this point, we part company with sequential modeles of computing, that is we leave the company of the batch processing. In exchange, the notion of "process" arises as the consequence of nteraction between continuing distinct computations, including possible interaction with the external world. For Milner and others this led to the development of process algebras. In the operating system world these notions as appear as processes, tasks, and threads.

Note that not just data has instances! Every invocation sets in motion a new instantiation of the effected unit of Computation, which then executes accordingly. Thus, while passively kept in units of Persistence uP(uC), the units of Computation (static program code) are actually meta-process descriptions of what is desired to be done at run time .[7]

10/5/2007© 2006, RJD

Units of Computation "come alive" when an executing process or thread passes through them as appropriate for the particular style of computation being used such as serial or parallel processing, and imperative, functional, or logic resolution execution environments. Most current computing equipment does this at a low level by manipulating the Program Counter which itself is an instance of a pointer to memory. Von Neumann architecture really does exploit the interchangeability of memory (units of Persistence) and computation. Unfortunately, most 3GL languages ruthlessly suppress that fact. Ironically, compilers and loaders could not work without it.[8]

Of course meta-processes and processes can be composed together as shown in *Figure 3: The Dynamic Composition of a Business Process.* It is important to see that meta-process composition is the not same as a "service." A service does something on your behalf, a meta-process composition tells you how to it for yourself, e.g. the difference between having a cake baked for you versus having the recipe to make the cake.[9]

Quite arguably, meta-levels model "abstraction[10]" and its companion "decomposition" much better than "encapsulation and interfaces." Nonetheless, in real world systems already built from multi-paradigm mixing, compromises become inevitable. Compositional Architecture can live with that. In fact, Compositional Architecture although presenting its own view of an ideal world, is actually the "Great Compromiser" through its willingness to compose anything with anything by any available means.

The ability to move feely between meta-levels in order to create and to manipulate knowledge traces its roots all the way back to Lady Lovelace who foresaw the power of computation in terms of the automated manipulation of symbols[11]. Compositional Architecture exploits the meta-data to data relationship and its counterpart, the meta-process to executing process relationship, as the powerful means to build high performance and yet highly flexible distributed systems in fruitful combination.[12] For examples of common architectural meta-models, see next *Figure 5: Examples of Common Architectural Meta-Models*.

10/5/2007© 2006, RJD

Figure 5: Examples of Common Architectural Meta-Models

OSI Meta Model Layers 1 thru 4 Architecture

Uses

OSI Transport Layer 4

OSI Network Layer 3

OSI Data Link Layer 2

OSI Physical Link Layer 1

Defines Minimum for

ANSI/SPARC Data Independence Architecture

User DBMS Views

Conceptual DBMS Schema

Internal DBMS Schema

Physical DBMS Data Organization

Describes and Defines

OMG 4-Layer UML Architecture

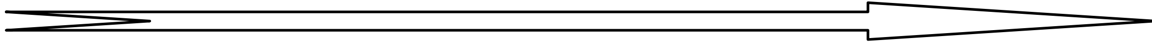
M3 **MetaMetamodel**
Meta Object Facility (MOF)

M2 Metamodel
CWM, UML Metamodel
e.g. Class, Interface, Attribute

M1 Model [1]
Specific UML model
e.g. Collection, SymbolTable, PurchareOrder, etc.

M0 "Objects" [2]
Specific Object Instances
e.g. specific PurchaseOrder number 4569105

1, 2 What many take to be the whole UML story!



Increasing Architectural Sophistication
from Conventional "Uses" Layered Architecture thru to Full Fledged Meta-Layered Architecture
In which each Meta-Layer also defines the Meta-Binding Rules (composition) for the Layer beneath it.

The Acquisition and Automation of Knowledge

Phillip Armour identifies software as the fifth great knowledge media (literally) since the beginning of time[13]:

DNA	Persistent knowledge store, updates slowly, little ability to intentionally design (by itself), effects outside world by building organisms
Brains	Volatile knowledge store, changes quickly, high ability to intentionally design, effects outside world through our bodies and by extension our tools.
Hardware/Physical Artifact	Persistent knowledge store, not usually easy to update, result of intentional design, effects outside world
Books/Writing	Persistent knowledge store that transfers well over space and time, slow to update, intentionally created, no capacity to (directly) change to the world
Software	Persistent knowledge store, quick to update, intentionally created, can effect outside world, but most all it is *active.*

The basic four entities of Compositional Architecture, namely the units of computation, distribution, persistence, and transduction, are seen as knowledge actors, knowledge containers, and knowledge distributors. The appropriate means of expressing this knowledge is relative to the problem domain at hand – ideally, the established concepts and notations of a given problem domain can be used directly, such as typically found for various science and engineering disciplines. The uniqueness of each domain's knowledge is why there never will be a one and only universal representation, nor a single all encompassing methodology for that matter.

Compositional Architecture stresses avoidance of domain knowledge contamination brought on by the inappropriate embedding and submerging of domain knowledge in the noisy details of implementation that will lose the core knowledge for any further use. For example, it is highly desirable to avoid embedding application level business rules into general purpose 3GL (C/C++/Java/C#) procedures, methods, and objects.

Within the software engineering domain per se, the usual suspects apply such as Requirements, Objects, Java, Class Diagrams, etc. Nearly all software engineering artifacts are overhead to structuring the mechanics (software and infrastructure) of the final system. In themselves, they have little or nothing to do with application knowledge content. This is similar to blueprints and piles of building materials which say little about the intended function (knowledge content) of a hospital.

10/5/2007© 2006, RJD

The principles of Compositional Architecture are compatible with Armour's "Levels of Ignorance[14]":

Oth Order of Ignorance (0OI)	Know enough about something (relevant) in order to use it effectively.
1st Order of Ignorance (1OI)	Know that that you don't know something.
2nd Order of Ignorance (2OI)	Don't know that you don't know something.
3rd Order of Ignorance (3OI)	Don't have a process to find what you were unaware that you didn't know.
4th Order of Ignorance (4OI)	Not even aware of the Orders of Ignorance and their implications.

Grave danger inherently lies in not knowing what we don't know. The goal is to move from higher levels of ignorance where the lack of knowledge is greatest towards the lowest level where all requisite knowledge is present. The process of reducing the levels of ignorance as well as removing noisy "unknowledge" applies across all meta-levels.

Armour sees what we don't know as the source of highest risk in any developmental cycle, generally even higher than external changes. "Ignorance" in the sense of what we don't know will cause the greatest variance in time and materials planning, and the danger that we will finally create a system that nobody wants. Consequently, the most important steps in system development are the identification of relevant questions rather the directly finding answers.

In a highly iterative methodology, ignorance discovery and knowledge resolution should be explicitly tracked. For each iteration, the status should be recorded of answers learned, unresolved questions outstanding, and newly discovered questions. We want to see progress made during each iteration on the unresolved questions while catching sight of any new problems that pop up. The "problem reporting system" often performs this function in more rigid non-iterative methodologies, although the danger exists that this will not be seen as a design tool.

The Six Fundamental Relationships

The Six Fundamental relationships within distributed systems are:

1. Network topology
2. Protocol Architecture
3. Binding: Transclusion and Metabinding
4. Open World Connectivity
5. Temporal Semantics
6. Security

10/5/2007© 2006, RJD

Network Topology

Network topology is widely studied in terms of directed graphs[15]. Network topology is the directed graph that defines which nodes directly link to which other nodes. In Compositional Architecture nodes are homes (temporarily or otherwise) to units of Computation and Persistence. Nodes also use the links, represented as edges in the abstract graph, to send and receive units of distribution ("messages").

Different architectural viewpoints and various levels of abstraction typically yield very different topologies because of the fundamentally different nature of the nodes and links at each level, e.g. servers using routers, word processors calling spell checkers, business rule engines composing rule sets from multiple sources of rules.[16]

Taking an example from basic TCP/IP architecture:

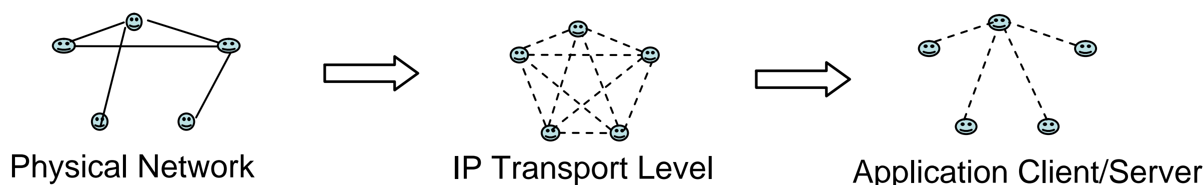


Figure 6: Topology Changes with the Level of Abstraction

Network topology is often relatively static, but it can be highly dynamic in self-defining and self-modifying environments such as encouraged by the process centric Pi Calculus. TCP/IP routing uses a relatively but not absolutely static network structure,[17] whereas the network formed by an instantiated "sea of objects" is highly dynamic in the extreme.

Protocols

A protocol defines the rules of engagement and exchange across links between communicating entities resident in nodes. Protocol design puts front and center the units of Distribution and Transduction ("messages"). This includes their associated abstract data semantics as determined by the units of Computation and/or Persistence contained within the unit of Distribution, or within the unit of Transduction on the internal digital side of the edge node. It also includes their asynchronous and isochronous temporal properties. Finally the protocol definition includes the partial order rules of permitted message sequencing which is also known as protocol "behavior".[18]

The principles of protocol design apply from the lowest transport level protocols (e.g. packets) to the highest application level protocols (e.g. banking transactions). Application Level Protocol Design is particularly friendly to amalgamation of Business Processes as protocols, Business Rules as transmissible Units of Computation, and with computer-to-computer "services," web or otherwise.

10/5/2007© 2006, RJD

Protocols are not directly the means to compose distributed systems, but they are the mechanism that enables such composition to take place[19]. As such, protocols are necessary "co-compositional" operators but they are never sufficient on their own. In effect, they eliminate or isolate the effects of network distribution so that composition of distributed knowledge content can take place when and where needed. Since protocols have so much to do with getting knowledge ready for composition and distribution, protocols typically define a large part of the distributed application semantics, in many cases even the largest part.

Protocol Centric Architecture (PCA) emphasizes the centrality of protocol specification in distributed system architecture. Similar concepts appear, at least in a more limited sense, in "Service" Oriented Architecture (SOA) and in even more limited form in "Web Services." Because protocols are only necessary but not sufficient on their own, Protocol Centric Architecture must fold into and forms a vital part of Compositional Architecture to which it ultimately belongs. For example, "Business Processes" may be treated as protocols, but they need "Business Rules' to be complete.[20]

Binding: Transclusion and Metabinding

Transclusion and Metabinding make Compositional Architecture highly suitable for distributed systems. Transclusion allows remote knowledge content to be acquired dynamically across the network and incorporated for immediate use locally. Tranclusion is the distributed version of "by value" for the four elemental types: uC, uP, uD, and uT, in which units of Computation and units of Transduction in particular also serve as "first class (abstract) data types." While early binding and late binding play a significant role in any distributed system, Compositional Architecture also emphasizes metabinding whereby an entity and its linkages come into existence dynamically, and typically on demand as needed[21]. Transclusion gathers the local and remote knowledge content for metabinding. Thus, transclusion is the "loader[22]" for metabinding. Important aspects of how Compositional Architecture differs fundamentally from Object Oriented Architecture is summarized below.

	Object Oriented Architecture	**Compositional Architecture**
Encapsulation	The most fundamental property of what makes an entity into an "object"	Rejected in favor of total information transparency for (re)composition as needed
Communication	Rigidly formatted Method invocations e.g. CORBA, COM+, ODP	General Messages: (uE & uT) e.g. XML, MIDI, MPEG, HTML, Javascript, XLST, and CORBA, COM+, ODP, etc.
Meta Level	Class Inheritance	Metabinding: "Generative" Compositional Rules
Instantiation	Subclasses and Objects	Transclusion and other forms of creating an entity
"Behavior" (Event Sequencing)	None	Protocols define allowable sequences of messages
Accommodation for other types of Architecture	Object Wrappers: the OO "hammer" makes everything to look like object-style "nails"	Use as is, where is: Compositional Architecture is the "Great Compromiser" and the "Great Integrator"

Figure 7: Comparison of Object Oriented and Compositional Architectures

10/5/2007© 2006, RJD

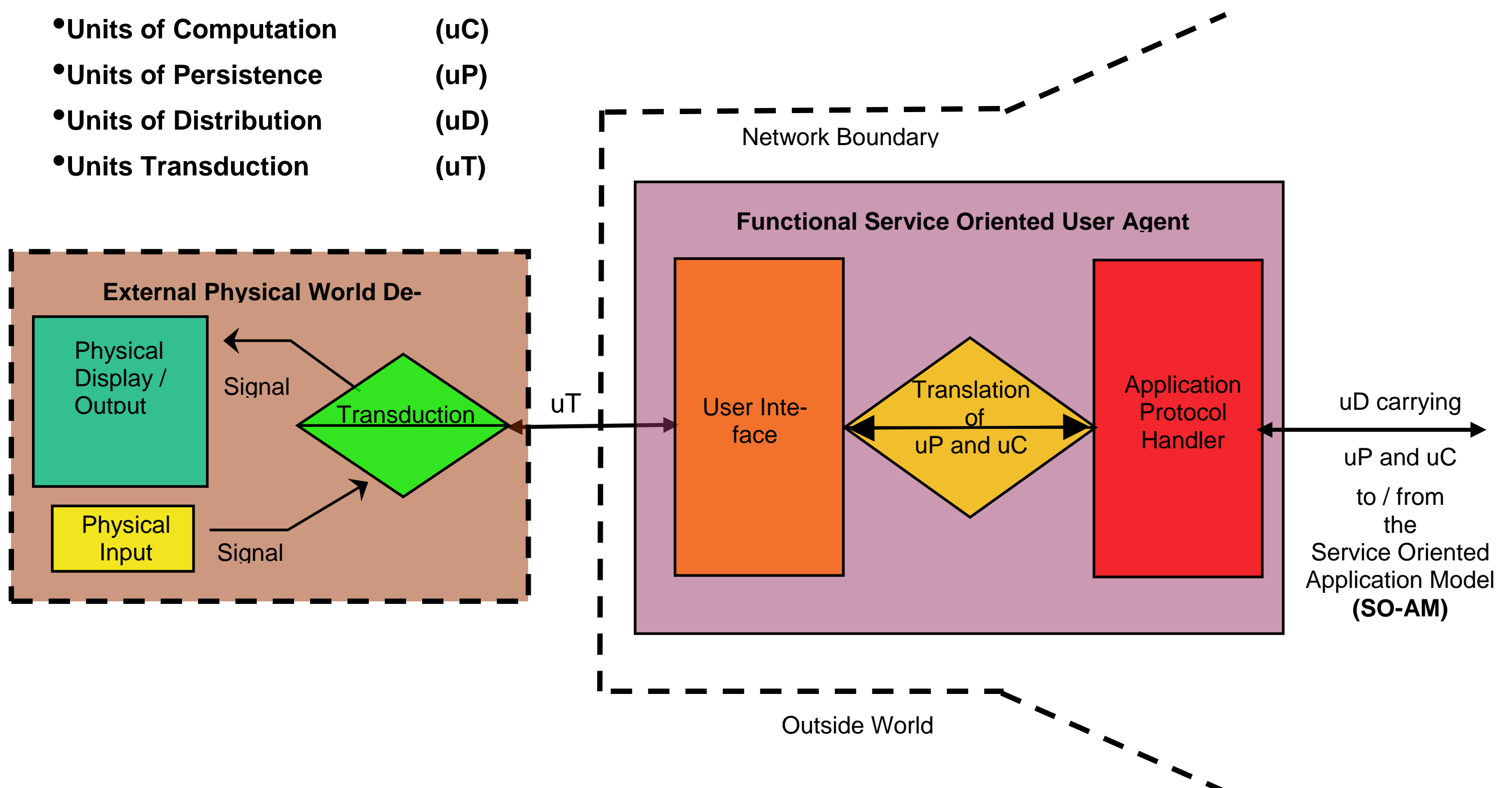
Open World Assumption

In Protocol Centric Architecture hence also in Compositional Architecture, user agents, being a special type of edge nodes, are seen to have two major aspects: the user agent represents the user to the network and represents the network to the user. In essence, a user agent mediates between the user and the network, specifically a user agent translates between the Application Level Protocol and the end user.[23]

The network facing side of the user agent speaks the appropriate network protocols to request work to be done on behalf of the user, which possibly spreads out widely within the distributed system, and speaks further network protocols to receive information in from the rest of the distributed system. The other major element of a user agent is the User Interface that directly controls interaction with the user. It makes pixels glow, voices sound, responds to mouse movements and the like. "Graphical User Interface" (GUI) functionality roughly corresponds to that of X-Windows, which correctly foresaw the local user interface software as a specialized kind of network server in its own right. Voice recognition and voice mail menus provide yet another form of User Interface.

AJAX clients have caused a great deal of turmoil precisely because they represent yet another swing of the pendulum regarding what a user agent should look like. They are the latest iteration of how a user agent should be created and distributed, where dynamically loaded units of Computation and Persistence should take place, and what is the appropriate application level content that should be received from and sent to them, as shown next.

Figure 8: The Functional Architecture of an Application Service Oriented User Agent (SO-UA)



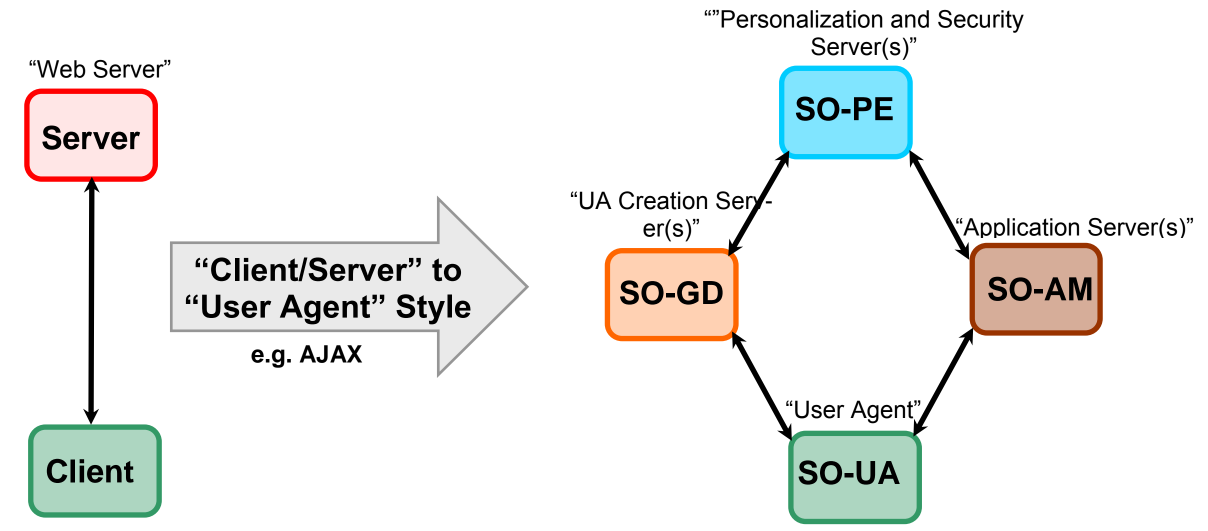
The real value in applying AJAX technologies is not "faster user response" nor "fewer round trips to the server." For that, they would hardly be worth the disruption of well-established, even if somewhat incoherent, techniques to construct web-based systems. Rather, Service Oriented User Agents, and their variants that may provision some application services locally within the User Agent, are a paradigm shift away from the modality of the WWW as hypertext. If anything, they imply the end of the "web server" as we know it. Let "client server" architecture finally pass on like the centralized "host architecture" of the prior mainframe era.

The architecture of Service Oriented User Agent follows the principles of Compositional Architecture by recognizing Service Oriented User Agents (SO-UA) plus three distinct server functionalities:

- SO-GD: Service Oriented UA Generation and Delivery - this class of server generates the user agent for delivery to and autonomous operation within the browser. SO-GDs exclusively serve up User Agents to the edge nodes that have a human presence.
-
- SO-AM: Service Oriented Application Model - this type of server strictly supplies application content without regard to or knowledge of its display, i.e. SO-AMs are pure application "model."

- SO-PE: Service Oriented Policy Enforcement using XACM/SAML terminology – this kind of server provides personalization and general security rules to the UA delivery and the application servers. SO-PEs never talk directly to the User Agent[24].

Figure 9: The Evolution of User Agent Architecture showing "Scientific" and "Common" Names



Service Oriented "Scientific Names"

- **SO-UA** **User Agent**
- **SO-GD** **Generation & Delivery of UA**
- **SO-AM** **Application Model**
- **SO-PE** **Policy Enforcement**

Note this diverges radically from JSP/ASP "Sever Page" Architectures and especially from J2EE Architecture – but it is "Java friendly" or to any other language(s) for that matter.

10/5/2007© 2006, RJD

There is a strict separation of concerns. Unfortunately the popular "Sever Page" architectures, JSP and ASP, fail miserably in this regard. They stumble further over the exceedingly inappropriate Model-View-Contoller (MVC) pattern. Despite being much hyped as a "positive" feature, the MVC pattern is to distributed systems including the Internet as the GOTO statement is to programming languages – a completely unnecessary source of deep muddle.[25] Instead SO-UA cleanly divide their attention between the display independent Application Level Protocol and interacting with the user via the currently selected and downloaded User Interface.

SO-UAs are services in their own right – they mediate on behalf of users with possibly more than one physical server on the back end – thus they provide yet another network "service" to the other nodes in the network. For the same application server they can provide multiple customized views differentiated by user or by type of users. They transform the meaning of "download[26]" from the joint fetching of application content and display content in one action, to two distinct actions: fetching User Agent content separately from fetching application content via the Application Level Protocol.

Most importantly, SO-UAs imply a deep transformation on the back end – they keep application servers from having anything whatsoever to do with the final presentation of application knowledge[27]. Thus instead of "web containers" we want to reach directly into application specific servers.[28]

Temporal Semantics

The meaning of time is one of the thorniest problems in meta-physics, and so in computing too. Time is also one of the most often overlooked and misunderstood issues in architectural design. For example, many people think that a "real time system" requires an arbitrary level of critical latency before which the system must respond to an event. This is a misleading notion. A real time system is any system that as an inherent part of its application semantics has an explicit notion of time as a fundamental parameter in computation and persistence called virtual time, and that notion of virtual time has a defined relationship to physical time (real time), e.g.

(1) $BORROW_BOOK_{VT} = 01/13/2009_{RT}$

(2) $OVERDUE_{VT} = BORROW_BOOK_{VT} + 14_DAYS_{VT} \rightarrow OVERDUE_{RT} = 01/27/2009_{RT}$.

Many, possibly most, systems thus have real time properties. In part, most protocols do as well, e.g. a defined protocol "time out" relationship TIMEOUT(START,DURATION) involves the matching of virtual time to real time similar to the library example above.

Compositional Architecture recognizes that distributed systems often, in fact nearly always, imply a notion of time to make sense of the multiple activities that go on. It is, however, out of scope here to venture further in the domain of temporal semantics beyond a brief mention of the most common aspects as they related to protocol design.[29]

10/5/2007© 2006, RJD

It is worth noting that "synchronous" and "asynchronous" are frequently overloaded terms in protocol design. In terms of the "behavioral" sequencing of messages, synchronous refers to a message pattern that requires a mandatory reply, while behaviorally asynchronous does not require a reply. Asynchronous protocols tend to be more useful for capturing the notion of causality (what triggers what) within a distributed system including "long running transactions," while being less concerned with master/slave or client/server issues. Note the meaning of synchronous and asynchronous in this sense in no way implies anything about the relationship to real time, e.g. in principle, the messages could be spaced apart by picoseconds or millennia.

On the other hand, the purely temporal semantics of synchronous means that the timing of events is constrained to happen at regular or predetermined intervals in real time, such as the sampling of continuously varying "analog" values, e.g. the digital sampling and transmission of multimedia waveforms. Conversely, the temporally asynchronous does not require any regularity or pre-assigned times. Time is an explicit parameter in synchronous communication whether referring to virtual time or real time. Thus, there is an explicit notion of the "clock" in the temporally synchronous, but not so in the temporally asynchronous. This parallels the difference between clock driven and event driven digital circuit design, which are the lowest level examples of the distinction. Although beyond the immediate discussion, this notion of "time as an application parameter" applies equally to units of Computation and units of Persistence as well as to units of Distribution, and units of Transduction inside of the distributed system.

Session Initiation Protocol (SIP) is the great hybrid that uses a behaviorally synchronous pattern, but temporally asynchronous, "call setup" phase to enable the subsequent isochronous data transfer such as VOIP. The temporally asynchronous call setup is the meta-level description of the forthcoming voice stream. The datagram nature of the Internet makes true isochronous protocols impossible, but explicit inclusion of virtual time within the protocol and having enough bandwidth to avoid unacceptable real time dropouts (by meeting "critical latency") allows for a rough approximation. This is why multi-media real time transmission on the Internet is always a dicey proposition.

Fetching music and video recordings can avoid the network bandwidth problem by completely downloading to local units of Persistence before playing back via plentiful local bandwidth. Effectively, this approach shifts the distributed systems properties from the external network to the local node's internal network of local storage devices, processors, and the high bandwidth of multiple local memory caches and busses. In the abstract, nothing really changes in regards to the information content, except for the physical manifestations of the archite[ ]ctural units that carry out the required functionality.

Security

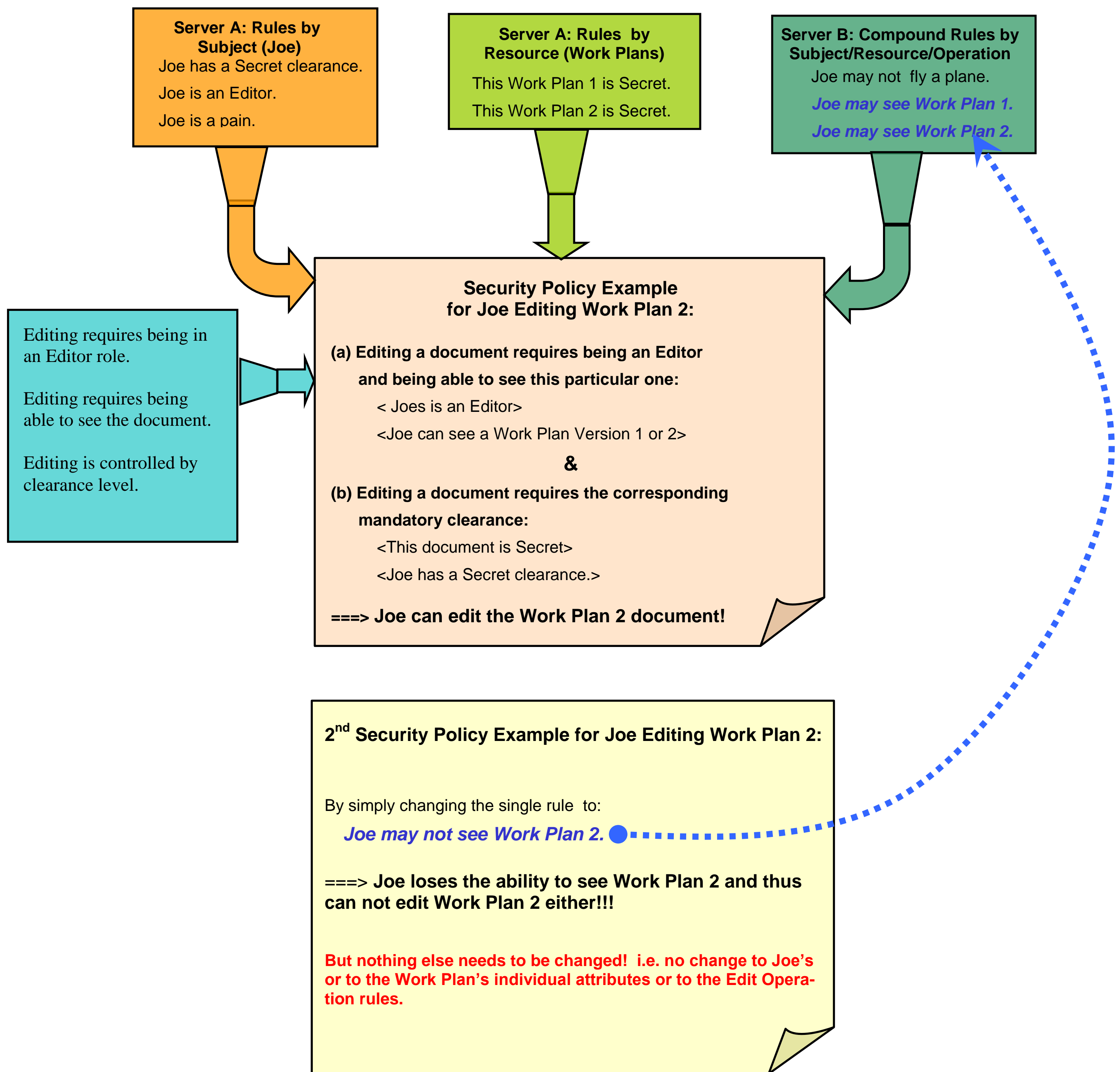
Security is a relationship. It is the relationship between what a distributed system can do and the limitations of what it should do. Compositional Architecture views security as a set of rules or equivalent formalism (security policy) composed with the rest of the application semantics:

$$\text{Total Application Semantics} = \sum \text{Functional Semantics} \;\&\; \sum \text{Security Semantics}$$ [30]

The effect of the semantic merger filters the permissible from the impermissible. The impermissible thus becomes the invisible. Generally, that which can not be seen can not be tampered with. Compositional Architecture fully supports the concept of defense in depth. Should knowledge somehow leak out regarding the existence of the intentionally hidden and hence forbidden content, the security mechanism can continue to block any attempt to make further use of the forbidden fruit.

10/5/2007© 2006, RJD

Figure 10: An Example of Security Policy as the Composition of Rule Sets



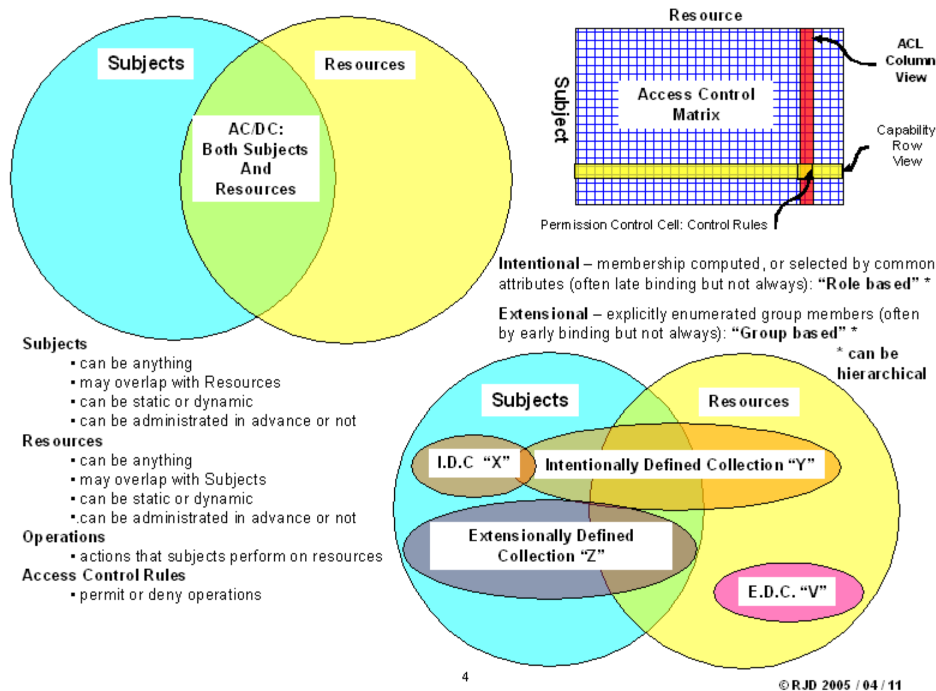
The Bifurcated Capability Model[31] (BCM) makes use of this notion of security in distributed systems for use with dynamically loaded clients, such as those built with AJAX technologies.

10/5/2007© 2006, RJD

Security semantics can be broken down along many dimensions. By tradition, the two dimensional "Access Control Matrix" places all the rules for a given resource in a single column and all the rules for a principle along a given row. Projection by column produces the Access Control List (ACL) for a resource and projection by row provides the "Capability[32]" for a given principle. Of course, additional dimensions could be added such as by operation and by temporal interval. Thus, the Access Control "Matrix" is really a hypercube[33] and a largely fictitious[34] one at that,

Groups of Principals "intentionally" defined as sharing common attributes such as "IT Manager" are often called "Roles." "Extensionally" enumerated groups such as the membership of a task force are often simply called "Groups."[35] Operations can also from hierarchies with inheritance like properties, e.g. a list of "file editing commands" that all share a common notion of doing something to a "file.

Figure 11: The Traditional Meaning of Roles, Groups and the Access Control Matrix



The exact meaning of a Resource, a Principal or Operation, literally depends upon the context of the application ontology in effect[36]. Groups can be sets of Resources, Principals, or Operations. Principals can also be resources, so that distinction is largely moot. Groups can be flat or hierarchical or any other kind of directed graph structure and can overlap. Resources, Principals, Operations and Groups can be dynamically created, adjusted, and destroyed. In any case, the applicable Group(s) rules (which can overlap with other groups' rules) have to be composed in with the appropriate individual Resource(s) rules (which can overlap) and the relevant individual Principal(s) rules (which can overlap).

Rules can and likely will collide at some point creating "non-monotonic" logic. These collisions have to be resolved in some manner to remove the contradiction between the rules. Generally, in discrete logics, one of the colliding rules pre-empts the other(s). The algorithm to do this, however, is open ended.[37] Other systems, such as neural nets, decide by means other than by the processing of discrete symbols.

10/5/2007© 2006, RJD

The Six Degree of Correctness

With all due respect to certain eastern traditions, functionally correct distributed systems under Compositional Architecture exhibit the following properties:

1.	Right Computation	(Right uC)
2.	Right Data	(Right uP)
3.	Right Place	(Right Topology/Node/Link)
4.	Right Time	(Right Temporal Semantics)
5.	Right Result	(Right uC/uP/uD/uT)
6.	Right Broadcast	(Right uD/uT/Protocol)

Of course, "right'" includes honoring all security constraints. It is left to the reader to determine how to do this during analysis, design, and implementation, and how to tell if you have done it correctly in verification and testing, but this lists what must be done.

The Scope of Compositional Architecture

Although simple in itself, compositional Architecture provides an integrative framework for a general model(s) of computing.

Compositional Architecture has more general use, including locally, than might first meet the eye. The Six Basic Building blocks, the Six Relationships, and Six Degrees of Correctness[38] describe an abstract version of any computing environment. These same notions re-appear at all levels of granularity right down to the wiring of the hardware. Applications arrange themselves into virtual interconnected networks just as surely as do routers and servers[39]. Inevitably, in an effort to avoid swamping the human mind with otherwise unfathomable detail, and to provide security mechanisms that wall off the permitted from the not permitted, the need for "modularization" of domain knowledge will create the distribution of knowledge and system architecture.

This ability to transcend levels of abstraction, especially for units of Computation and Persistence, is very useful to analyze and design large scale distributed data stores and application functionality at a high level, without getting lost in the endless details of data normalization, Object Oriented structures, and other often counter-intuitive lower level design artifacts. Any technique that claims to operate at the "architectural" level, rather than just software engineering in the large, must have it.

Compositional Architecture spans without replacing algorithm design, data analysis, complexity theory, digital circuit design, information theory, queuing theory, finite state automata, parallel processing, formal protocol definitions, graph theory, non-von Neumann architectures and all the rest of Computer Science and Software Engineering. Compositional Architecture co-features units of Computation, units of Persistence, units of Distribution, and especially units of Transduction. The inclusion of units of Transduction admits to an open world model of things external to the immediate computation such as hardware interrupts to drive software processes, real time sensors and controllers, multimedia, human interaction, and the like. In this sense, Compositional Architecture has physical as well as logical boundaries.

10/5/2007© 2006, RJD

The Limits: A Little Humility about Living on the Fault Line

No paradigm is complete without acknowledging its limits: Compositional Architecture enables dynamic distribution and composition but in itself does not define the semantics of the knowledge representation. Simply as an enabler, Compositional Architecture is strong enough to rattle the Object, Component, and Service Oriented models. Going beyond Compositional Architecture, we are long overdue for the architectural tectonic plates to let loose with the big 9.0 mega-thrust quake that takes architecture much deeper into the capture of application semantics: perhaps the semantic web or something a whole lot like it.

---

[1] The Pi Calculus holds another view in which Uc, uP, and uD all reduce to "processes." The only item of interest is the externally visible interaction between these otherwise sealed processes (uC&uP). But being a process algebra, these processes are composable across dynamically created and transmissible named (addressable) links uD(linkname). All computation involves the distribution of names and the dynamically co-joined mega-processes that result. Units of Transduction, however, are not reducible to processes in that their semantics lies outside of the computational framework. Admittedly, uT must be translatable on the inside (whether as processes, or uC and uP) but are not themselves the exclusive possessions of the of the distributed systems architecture. In principle, the Pi Calculus is Turing Complete, but then so is a Turing Machine. Actually, Pi Calculus is a significantly enhanced proper superset of Turing Machines. Although a very eloquent formal model of concurrent computation, taken alone the Pi Calculus likely would bog down in the same tar pit. Using the Pi Calculus to construct dynamic high level network relationships laid over elements (taken as "processes") composed from traditional modes of computing such as "services" has great merit, e.g. Business Process definitions in the manner of BPML and BPEL4WS. Nonetheless, the pure Pi Calculus framework still suffers from inherent lack of visibility into the knowledge contents within rather than just between these "processes." This might reduce the Pi Calculus to the role of composing protocols and protocol instances, albeit dynamically which, of course, is a most honorable thing in itself as discussed later in Protocols section.

[2] Spreadsheets provide an intriguing insight into Compositional Architecture, even though they are typically local in nature. Spreadsheets were the first "killer app" of personal computing precisely because they introduced a brand new model to mix computation (formulas) and persistence (cell contents). They did this right under the nose of the then young and aggressively emerging Relational Data Model. However, the spreadsheet model was so powerful even managers could use it, or overuse it as the case may be. Unfortunately, the spreadsheet's internal grid style network structure does not scale well nor lend itself particular well to distribution over data communications networks, i.e. spreadsheets do not make a good model for distributed systems architecture.

[3] Aspect Orient Programming (AOP) attempted to extend OO by emphasizing patterns that "cross cut" object classes, especially "inter-weaving" of originally independent aspects into new entities. The idea was sound, but the rigidities of encapsulation largely defeated it. In contrast, Compositional Architecture assumes inter-weaving, a.k.a. composition, to be the rule, not the exception. In addition, the use of meta-levels enhances to an even greater degree the "inventor's paradox:" it is often easier to solve the general problem than the special case in front of you.

[4] See *Figure 9: The Evolution of User Agent Architecture showing "Scientific" and "Common"* Names.

[5] Ontology is taken in its most general sense – ontology specifies the meaning of existence within a theory, the things assumed to exist within a theory, and the relationships between them.

[6] Ultimagely it gave rise to Milner's adoption of the all process ontology in the Pi Calculus: "Processes, nothing but processes, and all the processes!" Even in that kind of ontology, there are still meta-levels (which are processes) and notions of creating (instantiating) an entity so described (which are also processes). In general, the ontological struggle between process, function, and structure is an age-old philosophical debate that permeates much more than just computing formalisms. While it is beyond the scope of Compositional Architecture to settle such matters, depending upon the knowledge representation chosen, Compositional Architecture can make accommodation.

10/5/2007© 2006, RJD

[7] Executing a unit of Computation may be viewed as 'unwrapping" or "instantiating" it from its passive meta-process state inside of a unit of persistence to a newly active unit of computation, i.e. $uC_i(uP(uC_1)) \rightarrow uC_2$ where uC_1 is the meta-process, uC_i is a computation capable of instantiating the newly generated process uC2. As an alternative, instantiation can simply be considered a primitive operator, e.g. "new", "load and go" or "fork". Similar comments apply to the use of uP, uD, or uT as meta-level templates by means of a special instantiation operator in the style of OO classes, or by means of instantiation capable uC, e.g. composing instance data together with a Microsoft Report Description Language (RDL) template and rendering the resulting report.

[8] Units of Distribution and units of Transduction might be considered to exist only "on the wire", that is a message buffer content and the message bits flowing over the channel may be considered to be distinct. When buffered, uD and uT would be kept in units of Persistence as uP(uD) and uP(uT). Their actual transmission would have the property of instantiating a message for the purposes of sending or receiving by appropriate units of Computation, e.g. send(buffer) and receive(buffer).

[9] Another classic example being: "Give a hungry man a fish, and you feed him for a day. But teach him how to fish, and you feed him for a life time." Current ecological conditions not withstanding.

[10] This seems to be the logical evolution of Dijstra's "Levels of Abstraction."

[11] And now for the 21st Century: The DNA Challenge. Perhaps instead the of the Turing Machine being the sine qua non of universal computing, because we know it is not, the standard should be DNA computing. Such a standard would have to take multiple processes and multi-processing in stride, discrete state knowledge representation, protocols, but also virtual time and real time, continuous processes, stochastic processes, non-linearity, genetic algorithms, thermodynamics, quantum mechanics, and maybe sub-quantum mechanics such as string theory with its weird dimensionalities, geometries, and wildly non-linear interactions. This is one tall bill of material to fill, but in its defense, at least Compositional Architecture for its part does not get in the way.

[12] To stress a metaphor to the breaking point, units of Persistence might be viewed as forming "pages" out of the huge hypertext document consisting of the entire content of the World Wide Web as originally envisioned by Tim Berners-Lee. Units of Computation might be viewed as forming "programs" out of a similarly huge hyper-library of active elements that also spans the WWW. AJAX clients enable the move from "a page at a time" to something more like "a chapter at a time." Transactions of greater length, however, entirely defeat going any further with this metaphor.

[13] See *The Case for a New Business Model – Is software a product or a medium?,* Philip G. Armour, pp 19-22, Communications of the ACM, August 200/Vol. 43, No. 8.

[14] See *The Five Orders of Ignorance – Viewing software development as knowledge acquisition and ignorance reduction*, Philip G. Armour, pp 17-20, Communications of the ACM, October 2000/Vol.43, No. 10.

[15] Graph theory is one of the most fundamental formalism in all of Computer Science. It is hard to conceive of a single area of the discipline where graphs do not appear. Consequently, Compositional architecture rests on the firm foundations of graph theory and the Four Elemental Units: uC, uP, uD, and uT. Tremendous variation may occur in the mechanisms and formalisms chosen to be used at any one time, but it is hard to imagine that the basics of Compositional Architecture could not accommodate it.

[16] The question arises, do nodes, links and network topology represent knowledge? As often the case in architecture, the answer is: it depends - they might, they might not. If distribution is inherent in the application semantics, say the logistics of physical product transportation, the network may be very important to the knowledge representation. If distribution is just the accident of deployment, for example the location of a transaction within a server farm, then the network may represent little to no application knowledge content in itself. Just as the network varies by points of view and levels of abstraction, the value of the network representation varies accordingly.

[17] Traditional Internet services such as DNS, SMTP, SNMP, LDAP, employ a specific protocol for each service that implies a unique associated distributed memory model (uP), a distributed computational model (uC), and a distributed labeling scheme to find things (node, link and content addresses). This is true of even the basic delivery service provided by IP routing, in fact IP routing is one of the more complex, multi-protocol services with elaborate network wide data (e.g. routing tables), and algorithms (e.g. RIP, OSPF, IGRP). IP routing makes a clear distinction between routing protocols that exchange meta-data about the state of the network from the application protocols contents to be routed. See Traditional Internet Architecture: Protocols and Service on the Writings page at www.csgroup.com.

10/5/2007© 2006, RJD

[18] The instances of which are often said to form a (discrete system) notion of "time" as exemplified by the Language of Temporal Order Specification (LOTOS).

[19] The entities in communication continually send and receive transcluded content, thus a protocol may be viewed as a series of tranclusions. Each communicating entity must emit messages from itself, and make received messages a part of itself, repeatedly in legal sequence ("behavior"). "Overhead" messages are quickly discarded. The remaining content of interest kept locally represents the transclusion of relevant higher level content. This notion of message production and separation, followed by message reception and merger, has formal expression in Process Algebras as "synchronized" communication.

[20] Caveat Use Cases: In UML Use Cases tend to either enumerate functionality (units of Computation) or to specify protocol fragments ("scenarios"). The serious deficiency of this approach is that while the Use Case may capture a finite scenario, protocols actually define the infinite totality of all legal sequences, much like formal grammars to which they are closely related under directed graph and set theory. Both can be viewed as generative or as consuming of the utterances which are legal under the rules of the grammar or protocol. It is very, very dangerous to confuse the finite and the infinite in architecture. For example it would be highly undesirable to confuse a compiler that can parse and generate object code for just one program (a totality of one) versus a compiler that can parse and generate object code for the entire C language (an infinite totality of possible C programs).

[21] As Gosling noted, the Network Extensible Windows System (NEWS) extensively used the notion of "executable messages" in terms of dynamically delivered pieces of Post Script. Although not a commercial success, NEWS was in many ways a head of its time.

[22] Actually, by their very nature ALL loaders use transclusion either locally or remotely: linking loaders, interpreters, and class loaders included.

[23] This is the functional definition of a User Agent, although it may be architected horribly differently. Service Oriented User Agents (SO-UA) adhere closely to this division in responsibilities with a corresponding division in their architecture as shown in *Figure 8: The Functional Architecture of an Application Service Oriented User Agent (SO-UA).*

[24] This leads to the bifurcation in the Bi-Cap (Bifurcated Capability) Model.

[25] GOTO laden "Spaghetti code" has monumental perspicuity compared to the average JSP or ASP page. Unfortunately, the currently dominant "server page" architectures, JSP and ASP, both smear user agent issues in with application data, i.e. they entangle the functionality of SO-GD with SO-AM. Consequently, they spend an inordinate amount of energy trying to keep "server state" strictly synchronized with "browser state", e.g. STRUTS. The Model-View-Controller (MVC) pattern simply does not apply meaningfully in distributed contexts. Much grief has come of this, and the obvious reaction has been the sudden adoption of AJAX clients using technologies that in themselves are five to ten years old. But AJAX is only a collection of technologies, it can still be radically mis-architected by trying to keep the "bad old ways" alive, e.g. relying on MVC and continuing the unhealthy mixture of concerns on the server and clients sides. Instead, the much-ballyhooed problem of "state," e.g. "page flow," should be solved by the User Agent tracking naturally along with the state of the display independent Application Level Protocol (ALP). This is consistent with the notion that the User Agent translates the ALP on behalf of the end user.

[26] At first this may be particularly hard to grasp for conventional web programmers, especially the ASP and JSP subspecies. Remember, that weirdness like "page flow" does not belong on the server side.

[27] See Section 2- Protocol Translating User Agents and Models. As currently planned, the Architecture of Service Oriented User Agents is the follow on paper to this one.

[28] For example, SQL Server 2005 Report Services supports direct URI reference to pre-canned reports that bypass any web server by going straight into the report engine sitting on top of the database/data sources. Note that low level SQL is not used in this case. Internally, Report Services also has its own report definitions, the Report Description Language (RDL). RDL supports full notions of uC, uP, uD and uT from external sources via custom adapters. Microsoft marketing currently calls this concept "Connected Systems" – a most excellent expression.

10/5/2007© 2006, RJD

[29] Further discussion and examples of using Real and Virtual Time concepts architecturally can be found on the Writings page of www.csgroup.com, namely Telecomm Billing: Temporal Intervals & CA Principles and Telecomm Billing: Real Time and Virtual Time. Given its discrete mathematical roots, the Pi Calculus like all process algebras lacks an inherent sense of "clock" time (virtual or physical), instead substituting the weaker notion of partial order. Consequently, the Pi Calculus can not deal within its own framework with the continuous process aspects of applications such as signal processing, industrial distributed continuous process control, and the like. To do so would require an additional continuous/discrete (or analog/digital) framework, i.e. rules and operators for the analog-to-digital approximations of analog computation and persistence in the digital domain, and the digital-to-analog approximations of digital computation and persistence in the analog domain. Note that the value over time of a continuous memory differs drastically from a discrete memory that patiently holds onto the last value given to it – analog memories do change even when nothing else is acting on them! This complicates many assumptions (interchangeability and reduction) made about purely algebraically composed "time" such by LOTOS or the later Pi Calculus. Compositional Architecture and the Pi Calculus are certainly compatible, as are Petri nets, state machines and other formal techniques to describe discrete state re-active systems. Additionally, Compositional Architecture is compatible with other temporal formalisms capable of integer and real valued time, such as Allen's Temporal Intervals. As the saying goes – knowledge is time.

[30] The total rule set may be organized along other lines – the essential fact remains that the security semantics are an integral part of application semantics, not something foreign or to the side of it.

[31] "Bi-Cap" Documentation is under development.

[32] Capabilities align nicely in a principal oriented way with "personalization." Although the Bi-Cap Model (BCM) supports security rules organized along all dimensions, it works especially well with Capabilities. This is a huge leap forward from the near universal orientation in current systems towards resource based Access Control Lists (ACL). BCM allows much finer grained control over security rules. The Bi-Cap Model bridges security between co-operating but suspicious entities – SO-UA and SO-AM in particular – hence the full name Bifurcated Capability Model.

[33] Using multi-dimensional modeling borrowed for OLAP might lead to some interesting results at least in stating and analyzing security rules and policies, and possibly for organizing them into some sort of rule base.

[34] Many of these rules lie buried deeply inside of operating systems and subsystems such as databases and firewalls. This severely complicates the task of the Enterprise Security Architect who must use multiple mechanisms to enforce security policies. It also unfortunately tends to separate responsibility for security aspects from application analysis, design and development. This frequently results in the horses being long out of the barn before anybody vainly struggles to close the security doors and loopholes.

[35] See the preliminary Distributed Capability Security Model Working Notes at www.csgroup.com

[36] Note that the existence of a Resource or Principal does not necessary imply a unique "identity," such matters belong to the ontology. Similar comments apply to the meaning of Groups. The formalism chosen to fulfill the role of "rules" can also be deferred to the application ontology – strictly speaking, "rules" do not have to be formal logics, although they do have to be some form of computation (uC). Thus, they also can be "non-logical" things such as recursive functions, fuzzy rules, or neural nets.

[37] The reference implementation of XACML, for instance, comes with four different strategies and has the framework to add new ones as desired. Other rule strategies are also possible, such as fuzzy logic resolution.

[38] An interesting co-incidence of sixes.

[39] This is the great insight of the Pi Calculus and its "Process Oriented Architecture" (POA).

10/5/2007© 2006, RJD

Appendix A: Thoughts on Service Oriented Architecture as a Major Piece of the Puzzle

Architectural design focused on Application Level Protocol (ALP) specification has emerged (sort of) under the banner of "Service" Oriented Architecture (SOA) after decades of being eclipsed first by the emergence of the Relational Data Model and subsequently by Object Orientation. While Protocol Centric Architecture (PCA) has been a long, often lonely, time coming, it's arrival is most welcome. This development does more than just extend the prior picture of distributed system architecture – it forms a completely new kind of picture. Protocol Centric Architecture fully spans what is now called distributed OO, Service Oriented Architecture, "Business Processes" which arguably are specialized kinds of protocols, and "orchestration" in which the higher-level application protocol layers on top of lower levels protocols such as file transfers and "web services." Protocol Centric Architecture is a most disruptive technique that we will be mastering for years to come.

From a methodology perspective, Protocol Centric Architecture makes for "iteration friendly" development environments. Simply put, the application protocol suites provide the conceptual scaffolding upon which to build and to extend the nodes that host application computation and persistence. It makes sense to begin with protocol backbone and then expand out the nodal ribs.

Messages and Behaviors

In Protocol Centric Architecture, the activity happening on communications links define the architecture rather than the characteristics of the nodes. The design of the application level protocol may imply that persistent memory resides within the communicating nodes but the protocol design says nothing directly about how the persistence has to be achieved. Similarly, functional computations may be implied but their internal structure within a node is not specified thus they are not constrained in that dimension.

Protocol Centric Architecture differs from Object Orientation in that OO focuses primarily on local computations and local state while treating messages as side effects, whereas PCA views the messages to be primary and the local computations and persistent memory to be the side effects. The great glory of Protocol Centric Architecture is the necessity to specify only the meaning of how things are said, but never the nature of the things saying them, nor having to specify how they carry out their responsibilities. Thus network entities never anticipated by the protocol designers may make use of the protocol, much as programming solutions may be created to problems for which the original language designers never imaged.

Protocol centric design went by the name of "message passing architecture"[1] in early OO theory but that promise was never fully realized for reasons of loading all the other baggage onto mainstream OO. This was especially true because of the extensive over reliance on subroutine/RPC "method" interfaces, the implied memory models behind "attributes" and "class memory", plus the noisy anti-knowledge complications of inheritance. The concept of "data flow architecture" is closely related, but has to be augmented with notions of "process" as discussed earlier by means of persistence beyond the mechanism of simple function reduction.

10/5/2007© 2006, RJD

In terms of message passing, XML is extremely useful in many applications, but PCA can embrace whatever message optimization may be needed including the use of binary data techniques such as "C-style data structures", multi-media data types, caching, data compression, and even shucking the heavy weight overhead of TCP/IP transport protocols. PCA provides a drastic superset of whatever can be achieved by RPC based distributed OO/"Component" based systems. Happily, it does appear that what goes around does indeed come (back) around. Recognition and use of Protocol Centric Architecture is here to stay by whatever name it may be called.

In the rather Socratic discussion entitled "A Conversation with Roger Sessions and Terry Coatta" at the ACM Queue website, also known here-in as the "Coatta Dialogs," Roger Sessions offers the following observation:

"The big difference between Web services and CORBA is that the Web services people said right from the beginning: there is no API. The only thing that we standardize is how messages go from one system to another and the coordination around that. CORBA was 95 percent about how the client binds into the system. That was its downfall."

The Two Major Architectures of Protocols: Data Centric and Behaviorally Centric

From experience with Formal Description Techniques (FDT) such as the Language of Temporal Order Specification (LOTOS), it seems that protocols come in two basic architectural flavors: behaviorally oriented protocols and data oriented protocols. Either style of protocol can solve most problems, but generally, there are additional factors pushing in one direction or the other.

The first form features many short messages containing relatively precise operations with small amounts of associated data. This behavioral style of protocol design places much of its semantics in the multiple types of message operations. Sequences of messages ("behavior") carrying the numerous operators along with their associated parameters form the other major part of the protocol's semantics. Such protocols are behaviorally intense because they employ many different stock sequences ("behaviors") consisting of many different small sized messages going back and forth. Thus, interaction sequence ("behavior") also carries significant amounts of information. Not surprisingly, this is called the "behavioral" style of protocol architecture. The associated data in each message tend to do less of the semantic heavy lifting in comparison to their role in data intensive protocols. Obviously, the distributed OO and Component Oriented Architectures, and unfortunately the currently dominant "Web Services" architecture, favor this behaviorally intense style of protocol architecture. All of them implement the Remote Procedure Call (RPC) model as the primary form of message exchange.

10/5/2007© 2006, RJD

The alternative form of protocol design features relatively fewer but generally longer types of messages. The messages tend to have a single "op code" followed by extensive data that goes along with the requested operation. The bulk of protocol semantics lies in the data. This is the "data oriented" type of protocol architecture. It features relatively few exchanges of messages containing complex and lengthy data. Beyond some notion of "Service Access Point" (SAP) in OSI terminology, or "node" or "host" in Internet terminology, or named link in the Pi Calculus, these protocols tend to have little interest in how the requested operation is performed on the data and how the processing within the receiving node is structured. Certainly knowing about the "method" signatures of internal objects and components would result in a far too tight coupling between network elements. This in turn would create far too many unnecessary dependencies that would have to appear in the protocol's contract between communicating entities.[2]

Most traditional Internet services follow the data centric protocol model. For example, Simple Mail Transfer Protocol (SMTP) provides eMail transfer services on the Internet by means handful of requests that use a gaggle of complex headers and data types. The open-ended nature of the MIME data types single handily knocked out the competing X.400 eMail standard that once appeared poised to take over. DNS, X.500 and LDAP follow similar "data intensive" protocol architecture. Viewed through a protocol analyzer these basic Internet protocols often appear to be a conversation between two idiots talking to each other – but most importantly they are both machine and human readable[3].

The State of Protocols

A significant difference between objects and messages lies in the way that each may be persisted or transmitted, and then re-used. Both must be "serialized" because our data storage devices are operated and addressed serially. In serial state both must also be transmitted serially because our networks operate serially, and are addressed serially at least at the lowest levels (e.g. MAC and IP addresses). But serialized "passivated" objects prior to re-use have to be "re-hydrated" to the state they were in previously – including the tight binding of local computation and local persistence. A message can be processed any way that suits the receiving end regardless of how the sending end originally conceived of the message. The receiving end may think of the message as strings, as re-inflated DOM objects, parse tress, symbol tables, generic XML document, or in any other way that local software knows how to deal with the message, e.g. "message handlers" such as word document processors, eMail clients, and browsers. As such, messages and localized message handlers can support much richer data types, including executable rules and Turing complete languages, with much greater implementation freedom than objects can.

Much is made of so-called "stateful" versus "stateless" protocols. Strictly speaking all protocols beyond true send-and-forget datagram packets have state within themselves. A "response" to a request for which there is no record should be an alert that something is amiss as detected by Stateful Packet Inspection (SPI). For each node, the protocol's state determines which messages may be legally received next and which messages may be legally emitted next[4]. A node may be a dynamically spawned entity that participates in a specific session, such as TCP's IP address/port pairs[5]. In this case, state is at least implicit in knowing the remote addresses of the communicating entities.

10/5/2007© 2006, RJD

In practice, the difference between stateful and stateless is largely whether specific units of Persistence ("state") are considered to be implicitly part of the protocol or they are considered to be explicitly carried within the units of distribution (messages) employed by the protocol. To use a programming language analogy, the X-Windows API employs "encapsulated C" whereby "context" data structures get passed around explicitly between otherwise stateless subroutines. C++ differs by implicitly capturing the state within the object in the true OO style of local persistence. For robustness at protocol level, relatively stateless protocols have the advantage of being clearer by virtue of being more explicit, easier to implement[6], and more likely to steer clear of tainted OO practices. Stateless protocols also tend to have data oriented architectures.

Analogously, operating system processes and threads depend upon correct management of what Lampson called "state vectors" as the persistence of the process' current state[7]. Such operating system processes within a single processor[8] form nasty and largely unplanned pre-emptive co-routine style protocols between themselves as driven by asynchronously arriving hardware interrupts and other events within the processing environment[9]. Ultimately, the OS scheduler resolves (some of) this.

Who's Winning, Who's Losing and The Importance of Picking the Right Battle

In 1993 there were two would-be standards destined to battle for supremacy on the Internet. One was CORBA and its distributed object/component ilk that had virtually every major computer company in the world behind them. The other was a single-handed research project that scratched the itch of Tim Brenners-Lee. His notion of a world wide web built on hypertext used exceedingly simple requests, rich data types, and a genuine act of creative genius. His invention of the Uniform Resource Locator (URL) that later morphed into the Universal Resource Indicator (URI) rivals for our times the invention of the wheel in its profound simplicity and effect on global society.

HTTP/HTML with URLs quickly adopted the MIME and other data types to become the richest human oriented communications protocol on the Internet. More critically to the advent of Compositional Architecture, HTTP/HTML also quickly adopted computation as a downloadable data type, e.g. Javascript, VBScript, Flash. Of course in terms of data oriented protocol architecture, the World Wide Web's (WWW) richness lies exclusively in it data types, not in the embarrassingly small number of HTTP operators, and the small number of HTTP headers for that matter.

So, between CORBA and the WWW, we now know which one became "the web that ate the Internet" and which one is sputtering into obscurity and probable extinction. There's a lot to be said for data oriented protocol architecture. The wide spread acceptance of XML has drastically extended the power of being data centric in protocol design including the creation of specialized executable "ML" languages as embedded units of execution ("rules"). Most ironically, the continued embrace of the RPC model in the form of the SOAP protocol may be the most serious mistake made today by the "Web Services" community. Time to get over it; "method" calls are simply unnatural in distributed systems.

10/5/2007© 2006, RJD

Further on in the "Coatta Diaglogues", Session makes the insightful remark:

"When you look at Web services, you really need to categorize it into one of two types of applications: inter-enterprise or intra-enterprise. Google is an example of inter-enterprise.

My position has always been that inter-enterprise is a marginal area of Web services. It's the one that Microsoft and IBM peddle when they're talking to everybody about this. But the much more important area for Web services—the one that's being used many, many places—is getting different technology systems to interoperate within the same enterprise."

In other words, the computer industry yet again may have solved the wrong problem, used the wrong architectural model, and introduced a host of noisy anti-knowledge by means of extraneous design and implementation details. Most computer-to-computer applications, which are the target of Protocol Centric Architecture, do not require any more than a simple link and some simple XML to be exchanged. Sockets and XML have a lot to recommend for them. WSDL, SOAP, UDDI, (and MQ Series/JMS) sure make a lot of noise, but it is not clear they are always worth the pay back. Furthermore, as languages to implement "Service Access Points" (SAP) that process incoming and outgoing protocols without regard to any human interface, PERL and other scripting languages are very powerful. Even Java servlets may be useful as protocol implementations in a lower level language. However, as noted the same cannot be said of hybrids like JSP and ASP.

Building and Re-using Application Protocols

The specification of application specific protocols can and should begin very early in the design process, remembering that protocol specifications can be evolved iteratively as knowledge is gained through out the life cycle, including even during production[10]. Protocol definitions can first appear in the Functional Architecture[11], before the details of the final Technical and Application Architectures have been established – in fact, protocols specifications may be one of the few "artifacts" that survive the transition. Thus, Protocol Oriented Architecture actually spans all phases of architectural, developmental and system integration activities.

Responsibility for Protocol Centric Architecture naturally falls to the "Solution Architect," ideally in collaboration with problem domain Subject Matter Experts (SMEs). Later on, IT domain SMEs (DBAs, Technical Architects, Developers and Testers) will be drawn into the refinement of the Application Level Protocol. Given its early critical nature, obviously PCA can make or break the entire project "right from the get-go."

Protocol Centric architecture offers a far better opportunity to achieve re-use than the much-vaunted ability of OO to provide re-use. It was always the weakest pillar of the OO propaganda as there are features in OO that actively conspire against re-use. The complex mechanisms and close couplings within and between object classes and object instances did little in practice to advance re-use due to these entanglements within a run time sea of objects. In reality, OO programming shares little in common with modular programming, although the Component Architectures made a stab at restoring the balance with some limited measure of success through falling back to the mechanism of encapsulation.

10/5/2007© 2006, RJD

In contrast, the most effective re-use has come at the higher levels of abstraction such as object patterns, general IT architectural patterns, and application specific architectures. Specifications seem to work much better for re-use than implementations. Protocol Centric Architecture offers not only a high level of abstraction in itself but by concentrating on what is said rather than how it is done, PCA helps to buffer the application design from details of the implementation. Protocol Centric Architecture captures a "constructive" (executable) meta-level description of a distributed system's operational nature that transcends both abstract design and concrete expression in one mechanicsim.

---

[1] In many ways the saddest "false dawn" of true Protocol Centric Architecture first appeared in "message passing operating systems" originally demonstrated by University of Waterloo's TOTH operating system and other research projects such as JADE at the University of Calgary. The highly successful QNX Real Time Operating System (RTOS) continues onwards to this day in direct lineal descendent from TOTH. Perhaps the most sophisticated version of massage passing appeared in the Linda operating system's notion of LOTOS-like tuple exchange that added persistence to the rendezvous model. Tragically, the Unabomber understood Linda's significant far better than many of his victim's colleagues.

The false dawn of message passing was so sad precisely because how much was shown to be practical, but ignored in practice. Crude Novell style distributed file and print sharing competed successfully in the office automation space against QNX. More generally, the Relational Data Model, and especially OO propagandization, roared loudly enough to drown out the potentially more powerful techniques of Protocol Centric Design. But QNX amply demonstrates to this day that not only is message passing a powerful paradigm, but surprisingly enough, it powerfully supports real time intensive distributed applications (D-RTOS) – something for which no other architecture has been able to show an inherent advantage. The QNX model of local messages as in-memory data structures passed between processes/threads and their optimized light weight network transmission was light years ahead, and sadly still is, judging by the dominant models in UNIX/Linux and Microsoft Windows.

[2] This does not preclude the protocol from extracting and thus rendering "visible" the node's entire knowledge content as may be required. Afterall, PCA ia part of Compositional Architecture.

[3] For those who care, ASN.1 is simply evil!

[4] The concept of defining protocols in terms of the currently legal inward coming messages and the currently legal outward going messages *[argh, what is it called?]* is actually the dual of classic "state" machines which shows the same information on incoming and outgoing arcs from the node that corresponds to the current state.

[5] In TCP, the session name is the concatenation of two matched IP-address/port pairs!

[6] Humans tend to understand static "data" much more readily than active "behavior." Witness the agony of UML trying to capture behavior after having succeeded brilliantly to capture the static relationships within and between classes ("class diagrams"). It is no accident that the wide spread exchange of large XML documents has succeeded where RPCs basically failed.

[7] Computer theorist tend to hate this fact – but beware there is one very important data value that can not be "save and restored", real time itself! Time waits for no process, whether in the wait queue or the run queue. From this reality comes the challenges of OS scheduling, real time critical latency, and the like.

[8] This is within a single machine image. The effect gets even wilder in tightly coupled multi-processing environments where true independent processes are also interacting – one true process per physical processor to be exact.

10/5/2007© 2006, RJD

[9] The re-emergence of threads as a programmatic device is an amazing and dubious accomplishment of modern web and application servers. Threads, unlike software processes with separate memory spaces, lay bare much of this unplanned interaction. Thread programming is inherently error prone, dangerous, hard to test, hard to debug, and widely destructive beyond the immediate disorder created by a bug. Threads re-create the notion of re-entrant programming, whether serially re-entrant or fully re-entrant, that was never easy nor desirable. Thus, the re-appearance of threads in preference over Hoare Monitors and such programming constructs is even more surprising. The instability of many high volume websites speaks to the very real risk stemming from intensive uses of threads. Perhaps in part this stems from the fact that it took the Java community nearly ten years to finally realize in Java 1.4 that its thread model had serious design flaws from the beginning. With a little analysis and attention to first principles, these problems could have been identified right at the start circa the mid-90s. At any rate, in terms of protocol design, such strangely interacting pre-emptive concepts are best avoided in Application Level Protocol design.

[10] Changes made to the protocol definition once released into production, require special care and versioning.

[11] Application Level Protocol implementations make very good candidates for early iterations. Doing so guarantees that the system integration " phase" actually stretches over every iteration, and never arises as one huge horrendous event close to the end of development where it can reek maximum havoc.

10/5/2007© 2006, RJD

Appendix B: Consideration of Object Use and Abuse in the Architecture of Distributed Systems

	Object Oriented Architecture	Compositional Architecture
Encapsulation	The most fundamental property of what makes an entity into an "object"	Rejected in favor of total information transparency for (re)composition as needed
Communication	Rigidly formatted Method invocations e.g. CORBA, COM+, ODP	General Messages: (uE & uT) e.g. XML, MIDI, MPEG, HTML, Javascript, XLST, and CORBA, COM+, ODP, etc.
Meta Level	Class Inheritance	Metabinding: "Generative" Compositional Rules
Instantiation	Subclasses and Objects	Transclusion and other forms of creating an entity
"Behavior" (Event Sequencing)	None	Protocols define allowable sequences of messages
Accommodation for other types of Architecture	Object Wrappers: the OO "hammer" makes everything to look like object-style "nails"	Use as is, where is: Compositional Architecture is the "Great Compromiser" and the "Great Integrator"

Repeating *Figure 7: Comparison of Object Oriented and Compositional Architectures.*

In the Beginning and Thereafter

Since Parmas made the core observation in the early 1970s regarding encapsulation and information hiding, the theory and practice of software "objects" have progressed to ever greater complexity and diversity of application. Object support has become de rigueur for any new general purpose programming language, design methodology, or so-called Object Oriented (OO) analysis. Although OO has done many good things and a huge number of positive writings have appeared about the theory and practice of OO, this is not one of them. Indeed, it risks the sneers of the Computer Science and Software Engineering establishments to ask if the OO Empire has become over extended and threadbare.

Object Orientation has grown to encompass a number of properties while still including encapsulation at the very core. Polymorphism ranks not too far behind, and more controversially, some dated taxonomies include inheritance as fundamental. Together these make up the the "PIE" attributes: Polymorphism, Inheritance, and Encapsulation. Other features have been added to the OO model, including the Remote Procedure Call (RPC) model of distributed object invocation as found in CORBA, DCOM, JAVA RMI, and ISO ODP. Objects have been allowed to migrate in a few schemes, such as Java's ability to serialized and transmit objects, although in practice object migration is rare. OO has been extended with modest success in the direction of bulk persistence via OODBs.

10/5/2007© 2006, RJD

Elaborate higher levels of meta-design have been developed to describe OO systems, the most well known being UML. So called "Model Driven Architecture" (MDA) has enjoyed modest success applying OO to the outer limits of Software Engineering in-the-large. Of course it would be very hard to imagine an architecture that was not model driven in some sense, OO or otherwise.

But real architecture balances the issues of the problem domain with those of the solution domain, thus architecture extends well beyond Software Engineering per se. OO has been proposed as suitable for real word application analysis. But like most computer centric techniques, OO analysis has met with only limited success. As Philip Armour has observed, all of IT (Information Technology/Techniques) including analysis is ultimately about the ontology, epistemology, representation, and semantically meaningful automation of domain specific knowledge. A few application areas naturally come structured for analysis in object wrappings such as the software agents behind a physical User Interface (UI), but only a very few.

At the center all this OO expansion lies the fundamental concept of encapsulation. Indeed the emergence of "Component Architectures" and "Interfaces" as language elements gets back to the core notion of encapsulation. These can be seen as intentionally stripping away the Rube Goldberg artifices injected into later OO frameworks. But what is encapsulation?

Encapsulated objects are made up of two basic elements, computation and memory, neither of which is an object. The binding of internal data elements with internal computational elements composes the encapsulated object. These computational elements are the only ones with the responsibility and the permission to directly access the object's local data including transforming it or portraying it in a virtual manner to the outside world. Of course, other entities, meaning the computation elements in other objects, will participate in system wide data flows that ultimately effect the data as well - but only by going through the internal "guardian" computational elements (methods) that are the closest and most privileged in regards to the local data within the object instance.

For Parnas, encapsulation meant "information hiding" to keep the code's dirty little secrets private while showing by means of its interfaces the social responsibilities, later known as contracts, that the entity fulfills. Methodological considerations led to this approach. Information hiding served to protect relatively static portions of the design from turbulence created by design decisions likely to change[1]. The intricacies of Third Generation Languages[2] (3GL) including the C family of C/C++/Java/C# and related programming languages make this attractive. Such general-purpose implementation languages spend the bulk of their effort mostly doing things that have nothing to do with the problem at hand, except adding the extra details necessary to assist the computer in carrying out the desired functionality. In the limit, one reaches binary memory maps as the final representation (1GL machine languages).

These general-purpose programming languages are equally good at nothing in particular. But this leads to a surprisingly wide ranging equality of mediocre-but-good-enough-solutions. Other competitive architectures such as demand flow and data flow (function reduction) architectures and logic inference architectures have failed to achieve such generic flexibility. Perhaps because, they also were at too low a level[3]. As such the 3GL general purpose languages echo the advantage of the archaic but hard to replace Von Neumann architecture. In short, the internals of such objects are assumed to have only bulky anti-knowledge in Armour's terminology.

10/5/2007© 2006, RJD

Drowning the Baby in the Bath Water

Unfortunately, this emphasis on hiding implementation detail drowns the baby in the bath water. What real knowledge does exist internally becomes hidden away along with the noisy anti-knowledge details of the implementation language. All 3GL languages are notorious for "burying the application knowledge under a mountain of code", thereby complicating the maintenance, porting, evolution, and eventual replacement of legacy systems.

For OO this is no less so, and possibly more so. Nearly without exception, OO languages fail to adequately define the "behavioral" sequence of acceptable invocations, viz they lack elementary internal protocol support beyond a single request and response from one method at a time. This makes OO unsuitable as a means of expressing distributed services and protocols. Most OO languages and design techniques tend to be best at the static structural aspects that involve barely disguised use of semantics networks. The existence of these relationships at the meta level (class) are specified, but generally not their internal application semantics because that is obscured by "information hiding." Most if not all OO languages in the 3GL tradition specify an object's incoming requests via "method signatures" but fail to emphasize the object's equally important outgoing requests to other objects. That is very bad form from a network perspective. In terms of the responsibility within a network of objects or any network in general, this "input only" syntax is dangerously lopsided. OODBs tend towards being the network data model rehashed, which is not a bad thing, in itself,[4] but neither is it particularly distinctive. Actual instances of objects become embedded in an executing "sea of objects" for which there generally is no adequate representation at run time. And if there is, such as a GUI framework's notion of object managers controlling various composite object instantiations, it is completely unrepresented and orthogonal to the class structure expressed in the OO language.

Object data "attributes" are basically a distributed memory model overlaid somewhat unnaturally on top of the encapsulated object model. "Class memory" shared by the instances of a class exhibits an even more blatant shared global memory model, despite shared global memory being anathema to encapsulated objects. To twist a quote in keeping with Dijstra's famous dictum, OO is mainly syntactic sugar, but one that can cause malnutrition through systematic knowledge starvation.

Upon closer examination, OO can not define itself. OO never developed good means to express the totality of useful knowledge within an OO systems, in part because no such technique could be crammed into an OO framework[5]. You simply can't hide what you do and fully specify it at the same time. Furthermore, any language good enough to express the contracts between objects could not itself be Object Oriented without suffering from infinite regression. However, non-OO based Protocol Centric Architecture (PCA) can do precisely that last job.

The Object of Persistence

Objects, and "components" for that matter, have a problem if they wish to externally persist their data, or their units of Computation for that matter. Relatively unsuccessful OODBs aside, externalization of state is generally realized to create an "impedance mismatch" between the object's local persistent memory and long term external persistence, particularly in regards to Relational Databases. A similar, if not worse mismatch happens when trying to map object classes to document elements. In general, long-term external persistence is a problem largely handled outside of the OO paradigm[6].

10/5/2007© 2006, RJD

The internal binding of persistence and computation within an object may be a good idea, or a bad idea, depending upon the problem at hand. But in general, if IT's value proposition "payload" is seen as the definition, creation, and automation of domain knowledge, then at most object packaging is secondary. Furthermore, anything that gets in the way of the clear expression of knowledge content is inherently undesirable. The operationalized definition of "badness" thus includes undesirable hiding or obscuring of internal knowledge, along with the obfuscation of knowledge through noisy disruption from anti-knowledge. Unfortunately, the "booster stages" of our IT infrastructures, OO included, seem to be rather good at achieving orbital production at the cost of damaging the payload. Too often, we reach production by mangling the domain knowledge.

The Fine Art of Dynamic OO Composition

OO systems are link-oriented. They employ messages that carry data back and forth in rather rigid and brittle format according the method's signature[7]. Class and inheritance are also sophisticated mechanisms, but they exist nearly entirely at the meta-level. Most importantly, very few OO systems, and none widely used, allow the messages to carry computations across a link[8].

Since all computation is internally encapsulated, a traditional object literally would not know what to do with newly arriving computations, also known as "rules". That is to say, all OO systems support data flows after a fashion, but almost never accommodate "rule flow," i.e. messages may carry data but not computations.

Object linkage is really a weak form of composition. Any two or more objects can be composed together by their data interfaces, but in no other fundamental way, except perhaps shared "class" memory. Obviously, in distributed systems, class memory is of very limited utility. But what we really need in distributed system is more powerful means of composition.

This follows from the view that distributed systems contain knowledge from multiple sources that must be dynamically brought and composed together on-the-fly to perform useful work. Thus, by their very nature, distributed systems require that the messages must be equally capable of carrying computations from multiple source as they are capable of carrying data from multiple, although not necessarily the same, sources.

The relative failure of distributed OO systems such as CORBA and DCOM stems in no small way from the static nature of object definitions that predetermine the structurally inflexible composition of the internal computation with the internal persistent memory. At run time, contrary to what we really need, all computation and associated internal data for the object has to come from one place, namely from wherever the object instance is served up. "Component Architecture" shares the same flaw and does nothing to change the problem, except to succumb on an even larger scale. The central problem stems from reliance on rigid interfaces that ultimately trace their lineage back to the original structure of FORTRAN sub-routine CALL statement. Hopefully by now, we can do better in distributed system architecture than re-cycle the past glories of 1956[9].

The rapid success that XML has achieved gives proof positive that the more the level of knowledge representation moves up and away from internal computer perspectives such as objects and subroutines, the better off we will be. We can save Object Orientation for hand coding key portions of lower level "engines" of generic utility value such as business rule engines, application specific scripting languages, large-scale databases, and for raw manual code generation where essential.

10/5/2007© 2006, RJD

[1] Note that Parnas' concerns where at the meta-level of system specifications. Parnas assumed that we had good specification languages that have not yet emerged. Formal Description Techniques (FDT) such as LOTOS largely failed to gain a foothold. One alternative for protecting static design decisions from the volatile ones is to make them all dynamic and visible! In Compositional Architecture, we do not want to "abstract away" any domain knowledge – although we may organize it in different ways at different meta-levels. It is the meta-levels not the encapsulated modular "containers" that take on the role of abstraction." Parnas actually hinted at this approach by advocating that design begin with consideration of an application level Virtual Machine. The Four Fundamental Units of Compositional Architecture – uC, uP, uD, and uT had their conception years ago as part of design oriented around application specific Virtual Machines. They gestated for a long time until born out of necessity to understand distributed systems at a very high level. More recently, it became apparent that they apply equally at all levels.

[2] Machine language (1GL), assembly language (2GL), and imperative languages with formal grammars (3GL) which include the major OO languages.

[3] There is an interesting divergence in the meaning of "abstract" between Computer Science and Mathematics. In Computer Science, abstract means being closer to the domain knowledge and further away from the underlying computing mechanism. In Mathematics, abstract means more general, hence less detailed. These are opposites. They form a duality of the "abstract." To a mathematician, assembly language is more abstract than the C language because machine language contains the primitives from which C is derived - of course a Turing Machine is even more abstract and beloved! Perhaps in Computer Science, where abstract takes on application domain orientation, the real meaning of abstract is that we really don't understand the topic anyway! ☺ Bring in the Subject Matter Experts (SMEs)! In practice, Computer Science often flips back and forth between the two polarities. This usually happens without warning, depending on whether the discussion touches upon the more mathematically tinged considerations or the more application oriented ones. This paper makes no pretense of being any better than common practice in this regard.

[4] The dominance of the Relational Data Model and SQL as the query language results from fashion as much as any overwhelming inherent advantage. Semantically the "network" and "relational" database models are interchangeable. The relational model has quirks such as forcing explicit address linkage (foreign keys) where nameless connections in "owner-coupled sets" and their "extents" might be more natural, e.g. in a bill of materials and recursive relationships in general. It is interesting to note that the early hope for constructing distributed systems by means relational database replication has largely failed save in a few specialized applications. Instead, using XML and XPath can be seen as being much more intuitive data handling techniques for message passing than trying to exchange the contents of tables – ultimately water flows around a boulder, not matter how big.

[5] For example, most Object Oriented languages can express only instance of patterns, but not the pattern itself.
[6] This includes the persistence of the objects computational elements. While class libraries and "implementations" can be shared within a single memory issue, it is very hard do so on distributed basis. Hence CORBA fell back to sharing only "interfaces" but not implementations.

[7] The signature includes the method name following by the list of typed parameter in calling order.

[8] Ok, maybe Smalltalk. But in many ways Smalltalk is Lisp by another syntax. Djikstra would have been proud of how far the mighty eval() function got. Nonetheless, Smalltalk also seems to be fading in relative OO popularity.
[9] The first FORTRAN compiler actually predated the first symbolic assembly language by six months!

10/5/2007© 2006, RJD