Comparing Service-Oriented and Distributed Object Architectures

Seán Baker[1] and Simon Dobson[2]

[1] IONA Technologies plc, Dublin IE
sean.baker@iona.com
[2] School of Computer Science and Informatics, UCD Dublin IE
simon.dobson@ucd.ie

Abstract. Service-Oriented Architectures have been proposed as a replacement for the more established Distributed Object Architectures as a way of developing loosely-coupled distributed systems. While superficially similar, we argue that the two approaches exhibit a number of subtle differences that, taken together, lead to significant differences in terms of their large-scale software engineering properties such as the granularity of service, ease of composition and differentiation – properties that have a significant impact on the design and evolution of enterprise-scale systems. We further argue that some features of distributed objects are actually crucial to the integration tasks targeted by service-oriented architectures.

1 Introduction

Distributed Object Architectures (DOA) provide a stable computing platform on which to co-ordinate information systems within enterprises. The increasing desire to integrate very large, very loosely-coupled systems, and to automate business-to-business interactions, has led to an appreciation of the complexities of extending DOA technologies across enterprise boundaries. Service-Oriented Architectures (SOA) have been introduced with the goal of providing both intra- and inter-business services, and so potentially acts both as a complement to, and replacement for, DOA as an enterprise platform.

SOA and DOA are aimed at different problems, however. The origins of DOA lie with systems such as the CORBA standard, which was defined as a general-purpose integration technology. Because there was little support for implementing servers when CORBA was introduced in the early 1990s, CORBA implementations (and indeed the standard), concentrated on helping programmers to implement clients and servers. CORBA's commercial success made it a rival for other middleware: CORBA and the various other middlewares formed middleware islands, each with good internal integration, but poor inter-island integration. J2EE was introduced, among many other reasons, to provide a distributed object facility for Java.

SOA has been introduced to tackle the highest-level integration task, one important aspect of which is the integration of these middleware islands.

R. Meersman and Z. Tari (Eds.): CoopIS/DOA/ODBASE 2005, LNCS 3760, pp. 631–645, 2005.

DOA can be very specific to one specification or standard; SOA has to be more technology neutral. SOA has to work at a wider scale across an enterprise, across middleware islands in an enterprise, and across enterprises themselves.

One may debate, therefore, whether SOA is a minor re-adaptation of DOA ideas to a more XML-based world, a new departure in middleware that differs in fundamental ways from what came before it, or a business-driven replacement for old-style technology-driven middleware designs that better reflects modern concerns. This paper contributes to the debate by analysing the features that make SOA and DOA similar and identifies a number of subtle features that make them different. We take a more architectural perspective than that of Vogels[1], while reaching many similar conclusions. In particular, we argue that some of the features that some SOA advocates deprecate – especially standardised interface types and fine-grained object decomposition – might actually be key features in the integration task that SOA is targeting, while other features that are sometimes cited as being core differences – such as the use of messages rather than method calls – are of little deep significance. We argue that, despite being small individually, the sum of these differences leads to a radically different set of properties for enterprise-level system modeling and design.

Section 2 highlights the superficial similarities between the two approaches and argues that these mask the deeper differences which are explored in more detail in section 3. Section 4 concludes with some observations on how the features sets of DOA and SOA point towards an more complete approach to enterprise integration.

2 Similarities

For most of this paper we use Web Services and CORBA as prototypical examples of SOA and DOA respectively. This is simply to provide a concrete context for discussion, and should not be taken as suggesting that either system is a pure exemplar of the architectural style. The choice of Web Services carries some particular worries: it is very new technology and therefore not yet mature, and as it matures it may begin to concentrate too strongly on its own specifications and lose sight of the broader, technology-neutral requirements of SOA. We believe that, because SOA is used at such a high level within enterprises, it cannot be based solely on any one middleware.

Perhaps unsurprisingly there is no universally agreed definition of either SOA or DOA: however, there is a certain convergence on the general thrust of the two architectures:

Service Oriented Architecture. "A service is a set of functionality provided by one entity for the use of others. It is invoked through a software interface but with no constraints on how the functionality is implemented by the providing entity ... A service is opaque in that its implementation is hidden from the service consumer except for (1) the data model exposed through the published service interface and (2) any information included as metadata

to describe aspects of the service which are needed by service consumers to determine whether a given service is appropriate for the consumer's needs." (OASIS)

Distributed Objects Architecture. "[Distributed object] applications are composed of objects, individual units of running software that combine functionality and data, and that frequently (but not always) represent something in the real world. Typically, there are many instances of an object of a single type ... For each object type you define an interface. The interface is the syntax part of the contract that the server object offers to the clients that invoke it. Any client that wants to invoke an operation on the object must use this interface to specify the operation it wants to perform, and to marshal the arguments that it sends. When the invocation reaches the target object, the same interface definition is used there to unmarshal the arguments so that the object can perform the requested operation with them." (OMG)

These bare definitions obviously share many similarities, and one might argue (as indeed several commentators have) that SOA is simply a marketing reinvention of DOA. However, the similarities are actually rather deceptive, and we believe that some of the differences, although subtle, are vitally important in understanding the true relationship between the two approaches.

First the similarities. Both SOA and DOA are structured around remote entities (services or objects) which perform actions on behalf of clients. The remote entities typically export a strongly-typed interface defined using a language-neutral interface definition language (WSDL[2] or IDL[3]), transported using a language-neutral wire protocol (SOAP[4] or IIOP[3]). Not all implementations function in exactly this way: Java's J2EE architecture is based on distributed objects but has a language-specific interface language and wire protocol, although J2EE objects can also act as part of CORBA systems.

Perhaps the most common comparison between SOA and DOA contrasts SOA's technological neutrality against DOA's lack of flexibility. Web services are presented as being able to use lightweight HTTP interactions, but without being tied to HTTP or WSDL, and this is contrasted against CORBA's dependence on heavyweight IIOP and IDL. This comparison is largely specious, in principle if not always in practice. CORBA's interoperable object references (IORs) encapsulate multiple "profiles" for accessing the same object *via* different protocols, allowing an ORB that supports a number of protocols to access the object using whichever protocol is most appropriate. It would be perfectly possible for an ORB to use HTTP as an optimised lightweight transport while transparently interoperating with every other CORBA installation. Equally the dynamic stub and skeleton interfaces provide independence from IDL when required. (See [5] for more details.) The point here is not to praise CORBA unnecessarily but rather to focus the debate on actual rather than superficial differences between systems.

Both SOA and DOA provide invocation by clients of operations at remote sites. DOA "prefers" two-way interactions both because this matches the needs of a typical application and because CORBA poorly defines the semantics of one-way calls; and by default a method call invokes a remote operation and waits

for it to complete. SOA is more neutral, in that it supports interactions being constructed explicitly from messages. This makes one-way interactions simpler to construct.

Some authors have contended that this difference in invocation forms the essence of the difference between DOA and SOA. However, experience shows that this is deceptive. CORBA, for example, supports both synchronous and asynchronous communication. The former predominate, and normal use of IDL interfaces implies synchronous communication: however, one-way calls can be defined and more advanced RPC structures such as "promises" can be used to make two ways calls non-blocking[6]. By contrast, a large fraction of SOA systems make almost exclusive use of two-way synchronous calls constructed from messages. While DOA and SOA may have different "preferences", the similarities dominate[1]. While there *is* a difference between methods and messages it actually occurs elsewhere, not in the basic calling conventions – a point we return to in section 3.7.

SOA and DOA place similar emphasis on the signatures of operations, but place different emphases on interfaces types for the services themselves. DOA systems emphasise the use of interface compilers to construct client-side stubs for invoking methods. While some commentators have stressed that SOA operations can be called individually, without interface compilation, our experience in practice is that large-scale SOA systems are using interface compilers too.

Both SOA and DOA are essentially families of systems sharing common architectural principles, and it is instructive to see the extent to which these sets of principles also overlap. SOA advocates stress the separation of systems into services with well-defined interfaces accessed using a common communications framework, allowing composition of services and removing the boundaries between applications and middleware islands. DOA advocates would provide a remarkably similar list: indeed, CORBA's initial *raison d'être* was to provide such a bridge between different applications, later extended to operate between middlewares. The point of both architectures is to provide *interoperability* rather than *homogeneity.*

3 Differences

What, then, *are* the differences between SOA and DOA? The (alleged) novelty and dynamism of the former are often set against the (alleged) rigidity of the latter: it is hard to see how this can be true given their clear technical similarities. The source of these comparisons may actually reflect the differences in maturity between the two approaches: DOA has a small number of dominant implementatations, and the features and restrictions of these are used to define the overall approach. Even though SOA is too new now for such definition and restriction, the two have always have strong similarities.

However, perhaps the defining characteristic of enterprise-scale software architecture is its sensitivity to small differences, and we believe that SOA and

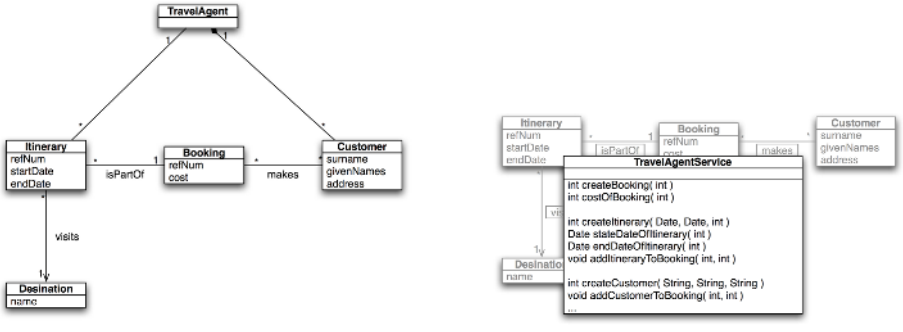
[1] It is certainly the case, however, that synchronous CORBA interactions are significantly more optimised that asynchronous calls in most ORBs.

DOA place this in high relief. Despite the similarities described above there *are* differences, and moreover *these differences count in aggregate*. A system engineered with SOA will be significantly different from one engineered with DOA, especially in terms of its long-term evolution. Moreover each approach has features that could be used beneficially by the other. It is these effects that we explore in this section.

3.1 Granularity

Granularity refers to the size of entities that are independently addressable within a system. In DOA systems these entities are individual objects; in SOA they are services. While this is a distinction that is, it must be admitted, impossibly subjective, we believe that some useful general observations may be made.

DOA inherits from standard (single-host) object-oriented programming a preference for fine-grained interfaces, although needing a somewhat coarser granularity to perform well over a network. Interface designers and programmers are encouraged to divide systems using interfaces providing a single abstraction (strong cohesion) and exhibit minimal dependence on the implementations of other interfaces (weak coupling). In designing a system for a travel agent, for example, a DOA designer might define interfaces for individual travel itineraries with methods for costing, booking and querying, and then make these accessible through an interface to a travel agent that collects together the itineraries and provides additional aggregate operations (figure 1(a)).



(a) DOA (b) SOA

(a) DOA (b) SOA

Fig. 1. Different granularities of visible objects

A SOA designer, by contrast, would probably take a more coarse-grained approach, for example designing a travel agent service providing operations on itineraries, with the itineraries themselves being specified by a reference number. The travel agent interface would provide itinerary creation, querying and so on, without exposing itineraries as an abstraction in their own right – although they would probably exist within the implementation (figure 1(b)). Indeed, the internal object model in the SOA case may be identical to that in the DOA case, differently exposed.

The difference is more than superficial. At a modeling level, DOA encourages an approach that is more purely object-oriented in the sense of identifying each abstraction and making it available in its own right as a first-class object. SOA encourages a decomposition in terms of real-world entities that have a concrete existence at the business level, and which captures the relationships and interactions between business entities without introducing other, extraneous abstractions.

As a SOA system evolves, new service interfaces will be introduced only when a new business entity is used. Continuing the example, a travel agent's interaction with a travel-insurance company will result in an insurance service interface being defined, again encapsulating the insurance business' business-level facilities. A DOA system evolves similarly, except that introducing an insurance service would also involve introducing a collection of smaller abstractions and interfaces (for policies, claims *etc*) that would typically be elided in public in the SOA case.

The coarse-grained, SOA view is that, from a business perspective, a travel agent is an entity "worthy" of an interface whilst itineraries are not: business interoperability should occur at the level of businesses, not with those businesses' internal abstractions. The finer-grained, DOA view is that itineraries, policies, claims *etc* are domain objects in their own right and should be modeled as such. This obviously reflects more that a simple difference in technology,. and cuts to the heart of the differences in approaches promoted by the different architectural styles.

There are significant advantages to the SOA view. It reduces the "surface area" of interfaces, and hence the learning curve for client programmers. It also tends to produce interfaces that perform operations in fewer interactions. The DOA view may lead to interactions that are too "chatty", in the sense of requiring a number of interactions to accomplish the same effect. In many systems this will increase the number of network operations to accomplish a single business-level task, reducing system throughput.

This may also explain why concurrency control and transactions are a significantly more visible issue in DOA than in SOA. A fine-grained system will inevitably involve multiple object interactions in a single business task, and so will need to make transactions visible to clients. SOA by contrast exposes the business tasks explicitly, and so is better able to abstract concurrency control behind the service interface. However the SOA approach needs concurrency control in order to scale, and these issues are re-appearing at the services level to support the emerging activity on web services transactions.

A significant literature has grown up around design patterns for object-oriented systems, some part of which now targets distributed object systems. (Good examples are [7, 8]). One immediate observation is that these design patterns almost always result in finer-grained decompositions of objects. This suggests that the usual object-oriented approaches translate well to DOA (with the caveat that they must be balanced against the need to avoid over-chatty interactions in the interests of performance and robustness), but not necessarily so readily into SOA.

3.2 Interface Types

DOA systems typically place significant emphasis on the definition of interface types. Interface types appear for applications (for example, standardised objects models for different vertical domains), for common services (for example the CORBA services and facilities definitions), and in the standards for DOA middleware themselves. The use of interfaces is pervasive and can be used to great effect: the CORBA trading service provides an interface for choosing between instances of an interface, with the interface repository providing a machine-readable description of the interface types.

SOA places less emphasis on common interface types: some commentators actually go further, asserting that SOA services should not have interface types that can have multiple instances[9]. However it seems inconceivable that a widely-deployed SOA infrastructure would not converge, at least to some extent, at least *ad hoc*, within specific vertical markets, on a set of commonly-agreed interface types. For example, a travel agency consortium could specify a shared definition and insist that all of its members adopt it. It is not sensible to prevent the consortium defining such an interface type – as indeed many are already – and the notion of SOA would be weakened if it does not actively support this.

Standardising interface types requires that there is an authority able to manage the standard. This authority may be defined in system-centric terms (all users of this system agree to these interfaces), or may be defined per-vertical (a set of common interfaces for the travel industry) or globally (a common interface to discovery services). In DOA there is typically strong agreement throughout a given system – although the system may be very large and most programmers may only be familiar with the interfaces of the sub-systems with which they work. In SOA by contrast – which targets even larger systems that DOA, and which specifically deals with systems spanning enterprises – there is no global attempt to control interface types.

Standardised interfaces within industry verticals can provide leverage at the modeling level as well. If we again consider the example from above, a travel agent using a travel insurance company will often be able to re-use the existing, agreed object model for insurance companies, reducing both the costs of extension and the degree of coupling between parties.

SOA advocates argue that the lack of standardised interface types increases flexibility and dynamism. However, interface standardisation evolved for a reason: without such interfaces, system developers must explicitly write adapters for each component. As system size and complexity increase, creating and (much more importantly) maintaining this "glue" code becomes the dominant cost. It is impossible to avoid the concern that large-scale SOA deployment may be better able than DOA to encompass a wider range of disparate systems, but only at the cost of hugely increasing the amount of interface adaptation and maintenance required.

In many large systems each interaction will typically only use a small fraction of each interface. This is perhaps more true for SOA than DOA, as the interfaces involved tend to be larger (section 3.1). One might argue that SOA adapters can

therefore focus on providing only the part of the interface required, which may simplify their development. Experience suggests that this argument is specious over the long term, as increasing complexity will tend to "fill out" the use of interfaces over time.

The reduced emphasis on interface type standards therefore does not remove the pain of standardisation for enterprises, but instead simply defers it to integration time and replicates it for each integration.

3.3 Composition

The usual response to an overly fine-grained decomposition (typically manifested as poor performance) is to re-engineer the system to coalesce several interfaces.

In many cases the internal design of a system will follow its external decomposition, with objects for itineraries *etc.* Coalescing interfaces will typically take the form of providing a façade that hides the individual objects.

Composition of services is an integral part of SOA, often referred to as *orchestration* and supported by a number of emerging standards (for example BPEL4WS[10]). Individual "partners" in a composite service are specified by providing the service port, optionally including a port type (interface type). Many programming languages have similar constructions, notably the structure and signature system in Standard ML[11].

One can provide orchestration without interface types. However, the absence of standardised interfaces make it difficult to see how a single process description can be re-used on different instances of services. The point is that interfaces provide more than individual operation signatures. An interface combines a set of operations with an implicit (or, increasingly, an explicit) contract on the way in which these operations will work together. Adherence to an agreed, reviewed and documented standard provides developers with confidence – albeit sometimes misplaced – that the operations will function together as intended and will respect the interface's contract. Allowing orchestration on the basis of individual operations, without this level of confidence in their consistent underlying assumptions, seems unlikely to succeed on a large scale.

3.4 Identifying Instances

Both SOA and DOA provide names for instances of interfaces: SOA typically identifies services by URLs for their service endpoints where queries should be directed, providing direct integration with the web, while DOA systems typically use more opaque identifiers such as CORBA IORs or J2EE object references.

A fine-grained DOA decomposition means that individual entities will typically have a distinct identity. We can identify an itinerary using an object reference, and both interact with it directly and pass it to other objects for them to use (figure 1(a)). By contrast a SOA decomposition focuses on service endpoints

which will not typically identify such a small object with a (SOA-level) identifier, and so will use an *ad hoc* reference such as itinerary number, reflecting the submerging of the object model behind the business interface.

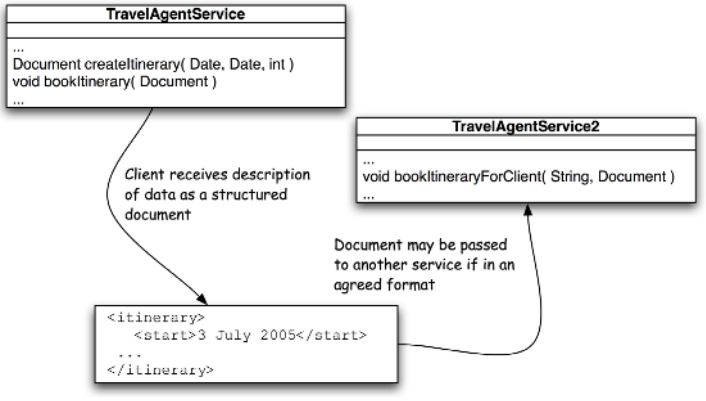


Fig. 2. Standardised documents replace standardised interfaces for data exchange

The coarse-grained approach has an immediate impact on systems architecture. Since data objects cannot be referred to directly by reference, the designer has two options. The first alternative is to identify internal data by some form of reference number, forcing the service implementor to manage an *ad hoc* namespace of objects. These namespaces are private, in the sense that an identifier generated by one instance cannot be used by another instance. To use the travel agent example again, a client must keep track of which travel agent service is managing which itinerary, so that queries are directed to the right interface. A service that used multiple services, for example when booking hotels through multiple providers, would be faced with quite a complex task. In SOA these issues must be managed outside the framework; in DOA they are typically managed by the middleware. This weakens SOA's claim to provide location transparency[12]: while technically true, the added complexity of namespace management reduces its impact in practice.

The second alternative (figure 2) is to manage data using a more document-centric approach, with a service creating documents that describe the data being manipulated: the travel agent provides an itinerary document in XML to the client, which may then store it or pass it back to the service at a later point. Such documents can be passed between services *if* they have an agreed format, so some level of standardisation across enterprises is again needed to facilitate integration. Services still need to convert documents to and from an internal representation and must take steps against alteration or forgery of date held externally, both of which complicate data management and interface coding.

If SOA is widely deployed, private namespaces and/or document formats will inevitably proliferate and complicate interactions. It seems possible to us that the DOA model therefore has significant advantages in terms of orchestration, service composition and re-purposing, as the single notion of object provides a significant simplification at the level of composition even though it may complicate individual interactions.

Some more structured approaches to endpoint description are emerging. BPEL4WS encourages the instanciation of business processes to handle stateful interactions, using correlation sets to identify specific instances[10–section 10]. WS-Addressing also provides more structure for addressing into individual services[13]. Neither approach provides support for the service implementor in managing their internal namespaces or entity lifecycles.

3.5 Modeling

Is the SOA or DOA modeling approach better? The answer essentially depends on the view we take as to what is happening when an analyst or modeler produces an object model. Both views may be derived depending on our initial assumptions:

If we take the view that the set of interface types identified are fundamentally tied to the system being modeled, capturing its essence, then it follows that these are unlikely to change unless there is a fundamental change made to the system. If such a fundamental change *is* made, any model – wherever it lies on the spectrum of granularity – will have to change. It also follows that such a set of interfaces will be easy to use because they model the real world so well. It *also* follow that making each object visible will allow improved re-use and reduce complexity, since the designer can focus on providing only the different functions in the system at each interface. We would therefore conclude that exposing the detailed object model will improve and simplify the long-term evolution of the system.

On the other hand, if we take the view that the object model is a detail in the day-to-day internal running of a business, it follows that the model will be subject to frequent change as the business evolves. It thus follows that such change should be masked from clients as it has no real significance to business-to-business interactions; would lead to undue coupling between businesses; and would anyway be too complicated for an external client to understand as understanding the decomposition (and hence the functions available) would involve too much understanding of the business. (This is essentially a model-level version of the well-known "yoyo problem"[14] at the code level.) We would therefore conclude that exposing only business-level functions, preferably in one interface and with no visible dependencies on other services, will improve the robustness of business-to-business interactions in the face of evolution.

Both positions are perfectly defensible. Our own view is that SOA cannot be dogmatic on the approach used. The guidance from practice would be that mainstream object-oriented decomposition can lead to over-fine interfaces for DOA. Interfaces cannot be so fine-grained that they will perform well only over very

fast networks: this would lead to too tight a coupling between a service and its clients. However, the SOA principles themselves – and certainly the technology used to implement them – should not restrict the modeling approach that an enterprise believes in. In particular, it is too fine a point for an architecture to *regulate* on how granular a set of interfaces should be.

Objects can be business level or system level. Viewed simplistically, services must be business level: however, it is difficult to draw the dividing line between business- and system-level entities, and this raises much the same issues around modeling as above. In addition, the principles behind SOA (such as well-defined contracts and separating implementation and interface) are just as valid at the system level as they are at the business level. (Some commentators even use the term SOA for the system level, and the term B-SOA for the business level; we prefer to define SOA as the uppermost level of integration and use the term Enterprise Service Bus to refer to the technology that facilitates communication between SOA entities).

In both SOA and DOA, it is common for programmers to generate service/object interfaces from some form of interface on existing systems (perhaps a description of the data that is exchanged, or interfaces defined in programming languages). These lead to low-level – and often inappropriate – interfaces. The only excuse for doing this in DOA is as a stepping stone to offering a proper high level interface to clients; there is probably no excuse for doing it in SOA.

The rise in interest in model-driven architecture[15] provides a possible bridge between the modeling views. MDA defines a system architecture as a set of transformations between models to reflect the different levels of concern in systems development. Coarse-grained SOA interfaces fit well into this scheme, although this still leaves the problems (exemplified in section 3.4) that arise from the inability to share model objects across enterprises.

3.6 Inheritance and Implementation

Inheritance is often regarded as a core characteristic of object-oriented systems [16]. Many DOA systems support inheritance at two distinct but related levels. At the interface level, an interface may specialise another by adding new operations. At the implementation level, it is often possible (although not required) to re-use the implementation of one interface in defining the specialisations of that interface.

Not all DOA systems have this feature, however. J2EE is particularly restricted in this regard, in that there is exactly one implementation of each interface within a single application instance, and that implementation is (somewhat perversely) not a valid Java-level instance of its own external interface type. The lack of polymorphism means that many approaches used in other DOA systems do not translate well into J2EE. It also complicates some simple but highly effective optimisations, most notably using language-level references instead of external network references when exchanging references to objects within the same server.

WSDL does not provide the notion of interface specialisation, perhaps in keeping with SOA's reduced emphasis on interface types altogether[2]. However, sub-interfaces provide a vital mechanism for differentiating providers without compromising interoperability: a client may use additional features provided by a particular provider if it understands them, but may rely on a core of operations being available uniformly regardless of provider. Run-time interface manipulations and sub-interface checks make this process reasonably straightforward. We persist with our view that interface types are actually *more* important for SOA than for DOA, given the focus on cross-enterprise integration.

Inheritance is not the only form of extension, however. Delegation-based models are also extremely powerful, and neither SOA nor DOA typically support the notion intrinsically. While such mechanisms are useful linguistically, they can be approximately provided using events (see below).

3.7 Operations, Messages and Events

Returning to the use of messages *versus* operations, one may ask the question *why* this difference is so insignificant, given the widespread belief that messages (in one form or another) are an essential feature of loosely-coupled systems.

Both SOA and DOA allow clients to invoke operations on individual interfaces. While messages (or one-way methods) may decouple the request from its completion, they do not change this model.

By contrast, event-based systems allow information to be transferred to zero, one or more consumers according to several possible models. In the CORBA event service's "push" model, receipt of an event automatically triggers the execution of handler code, so a single event can cause code to be invoked on the *different* objects that are subscribed to the *same* event channel.

The key difference is not (as is sometimes stated) between messages and method calls as invocation mechanisms, but rather between first-class events which may be manipulated programmatically and second-class invocation mechanisms that operate beneath the language level. To use the terminology of aspect-oriented programming[17], events reify the calling mechanism into the language and allow it to be modified to (for example) queue invocations or distribute them more widely. The XML nature of SOAP messages makes them easier to manipulate programmatically. Such manipulations are also common in CORBA systems however, albeit with considerably more effort being required.

Both SOA and DOA infrastructures have external event services (often with a standardised interface type in the case of DOA). Clients may interact with such event services using either messages or method calls, which are then converted into events and operated on using filters, publish/subscribe *etc.* While events provide powerful support for loose coupling, SOA's explicitly message-based invocation mechanism provides little or no such advantage over DOA.

[2] Although sub-interfaces may be added to later versions.

4 Conclusion

We have explored the similarities and differences between the service-oriented and distributed object approaches to enterprise system modeling, design and implementation. While the approaches (and their underlying technologies) share significant features in common, their differences – while subtle – lead to significantly different views of enterprise system structuring and evolution. This strongly suggests that SOA in particular represents more than simply a marketing phenomenon, and conversely that DOA has both a significant on-going niche and a number of important lessons to impart.

Of all the issues examined, four stand out. Firstly, SOA requires a significantly coarser granularity of exposed object model than DOA. The focus on coarse-grained, aggregated interfaces may simplify interactions across enterprises (or business divisions) by reducing the number of interface interactions needing to be understood. This is an important simplification in a world in which businesses interact more dynamically and without necessarily establishing long-term relationships between their information systems: programmers need to understand less of the target businesses' infrastructure in order to avail of its services.

Secondly, the ability to exchange references to objects within the framework seems to be a positive step for orchestration, allowing different providers to work with each others' data directly rather than *via* private references or descriptive documents – which require standardisation anyway. Forcing designers to deal with these additional complexities seems to serve no useful purpose.

Thirdly, SOA's de-emphasising of interface types will not lead to simpler integration over the long term. While simpler to establish, such *ad hoc* approaches push complexity out into each integration, the costs of which will spiral as SOA increases in market penetration. It would seem prudent to repeat the OMG's experience in defining standardised object models and interfaces within vertical market segments, allowing providers to differentiate themselves in other ways or by providing extensions to the basic structures.

Finally, business integration did not start with SOA. There is a existing volume of work in modeling the issues, objects, operations and relationships of specific industries. The significance of this work lies in the understanding it gives of industries and the degree of commonality that exists between providers, rather than in the object models *per se*. Perhaps, then, a more appropriate, business-level question is: how can this understanding be re-used in the slightly different context of SOA? It may be that the underlying modeling assumptions are too different to allow simple translations, although providing a service-oriented façade might allow SOA to use the existing DOA infrastructures with little additional cost.

These are extremely subtle issues, but taken together the different solutions that SOA and DOA take to them aggregate to deliver distinctively different system architectures. SOA systems are likely to use more asynchronous invocations and to involve less up-front cost to establish, although the complexities of adaptation may over time may make this a rather Pyrrhic victory. DOA systems may

more closely reflect a conceptual object model of the application domain and make information sharing and exchange simpler, but only in situations where standard object models and interface definitions can be agreed upon and when the value of the expected interactions justifies the standardisation costs.

The messages *versus* remote method calls debate misses the point. The distinction is rather between first- and second-class invocation mechanisms, and flexible event systems are equally definable in – and complementary to – both SOA and DOA, and so do not provide a reason to choose one over the other. Indeed, Gartner Group have suggested that *any* system architecture requires both invocation-based and event-based interactions to maximise loose coupling.

SOA's goals of a more "business-friendly" distributed platform are laudable, economically significant and technologically challenging. It is however important to remember and re-use the features from more traditional DOA approaches that can usefully be included into the top-level integration of enterprise systems.

References

1. Vogels, W.: Web services are not distributed objects. IEEE Internet Computing **7** (2003) 59–66
2. Christensen, E., Curbera, F., Meredith, G., Weerawarana, S.: Web Services Description Language 1.1. Technical report, World Wide Web Consortium (2001)
3. Henning, M., Vinoski, S.: Advanced CORBA programming with C++. Addison Wesley (1999)
4. Winer, D.: XML-RPC specification. http://www.xmlrpc.com/spec (1999)
5. : Common Object Request Broker Architecture (CORBA/IIOP), v.3.0.3. Technical Report formal/2004-03-12, Object Management Group (2004)
6. Liskov, B., Shrira, L.: Promises: linguistic support for efficient asynchronous procedure calls in distributed systems. In: Proceedings of the ACM SIGPLAN conference on Programming Language Design and Implementation, PLDI'88, ACM Press (1988) 260–267
7. Mowbray, T., Malveau, R.: CORBA design patterns. Wiley (1997)
8. Fowler, M.: Patterns of enterprise application architecture. Addison Wesley (2003)
9. Webber, J.: Horses for courses: services, object and loose coupling. Web Services Journal **4** (2004)
10. Andrews, T., Curbera, F., Dholakia, H., Goland, Y., Klein, J., Leymann, F., Liu, K., Roller, D., Smith, D., Thatte, S., Trickovic, I., Weerewarana, S.: Business Process Execution Language for Web Services, version 1.1. Technical report, IBM (2003)
11. Milner, R., Tofte, M., Harper, R., MacQueen, D.: The definition of Standard ML (revised). MIT Press (1997)
12. Waldo, J., Wyant, G., Wollrath, A., Kendall, S.: A note on distributed computing. In Vitek, J., Tschudin, C., eds.: Mobile object systems: towards the programmable Internet. Springer-Verlag (1997) 49–64
13. Box, D., Christensen, E., Curbera, F., Ferguson, D., Frey, J., Hadley, M., Kaler, C., Langworthy, D., Leymann, F., Lovering, B., Lucco, S., Millet, S., Mukhi, N., Nottingham, M., Orchard, D., Shewchuk, J., Sindambiwe, E., Storey, T., Weerawarana, S., Winkler, S.: Web services addressing (WS-Addressing). W3C member submission (2004)

14. Taenzer, D., Ganti, M., , Podar, S.: Object-oriented software reuse: the yoyo problem. Journal of Object-Oriented Programming **2** (1989) 30–35
15. Frankel, D.: Model driven architecture: applying MDA to enterprise computing. Wiley (2003)
16. Cardelli, L., Wegner, P.: On understanding types, data abstraction and polymorphism. ACM Computing Surveys (1985) 471–522
17. Kiczales, G., des Rivières, J., Bobrow, D.: The art of the metaobject protocol. MIT Press (1991)