Malleability, Obliviousness and Aspects for Broadcast Service Attachment

William Harrison

Department of Computer Science
Trinity College
Dublin 2, Ireland*
(+353) 1-896 8556
Bill.Harrison@cs.tcd.ie

Abstract

An important characteristic of Service-Oriented Architectures is that clients do not depend on the service implementation's internal assignment of methods to objects. It is perhaps the most important technical characteristic that differentiates them from more common object-oriented solutions. This characteristic makes clients and services malleable, allowing them to be rearranged at run-time as circumstances change. That improvement in malleability is impaired by requiring clients to direct service requests to particular services. Ideally, the clients are totally oblivious to the service structure, as they are to aspect structure in aspect-oriented software. Removing knowledge of a method implementation's location, whether in object or service, requires re-defining the boundary line between programming language and middleware, making clearer specification of dependence on protocols, and bringing the transaction-like concept of failure scopes into language semantics as well. This paper explores consequences and advantages of a transition from object-request brokering to service-request brokering, including the potential to improve our ability to write more parallel software.

Categories and Subject Descriptors D.3.3 [**Programming Languages**]: Language Constructs and Features – *abstract data types, concurrent programming structures.* D.2.11 [**Software Engineering**]: Software Architectures – *data abstraction, languages.*

General Terms Design, Languages.

Keywords Service-Oriented, Aspect-Oriented, Programming Language, Middleware, Concurrency.

1. Introduction

With current approaches to software design and implementation, software artifacts, like classes or methods, embody many decisions made at they time they are designed and implemented. In more fluid environments, including distributed, autonomic, grid, and service-oriented, that are emerging today, we need more of these choices to be deferred until run-time. For example, today the client of a method specifies where to find its implementation, whether in an object or in a service. In common usage, malleability is the ability of an article to be molded or shaped to fit for changing circumstances, and we have applied the term to software artifacts [5]. Enhancing malleability requires us to re-think the boundary between programming languages and middleware and introduce a point at which intelligent choice can be injected into the otherwise rigid semantic specification. But introducing a locus for such an intelligence allows us to better address the need for greater parallelism that we confront in the multi-core future.

To address the need for greater malleability, we advocate the use of a programming model, called the *broadcast service model*, with several novel characteristics:

- There is a modularizing construct, called *service*, that contains a coherent collection of classes and has a run-time instantiation. Services may be responsible for handling method calls made by a client or may be attached obliviously, as aspects. They may be bound within a process or located remotely, and they hold the state for various aspects of objects.
- There is an interface-like construct, called *face*, that characterizes a set of methods that can be called safely, but does not indicate which object or service implements them.
- Clients may use a single reference to an object even when its state is distributed across several services. The services are responsible for resolving references, so that an object's methods can access its state.
- Method invocations do not indicate a particular target object or service. Instead, invocation is broadcast, with the intelligence guiding the delivery of a method call to one or more implementations being provided by a middleware-defined dispatcher, caller a *service request broker*.
- All execution takes place within a transaction, which serves to circumscribe behavior on failure.

To explore this model, we are developing a programming language, Continuum [14], which embodies this structure and introduces constructs that enhance malleability. In the next section of this paper, we use a small example to illustrate the underlying issues of malleability and obliviousness, point-to-point service provision, aspect attachment, and broadcast service provision. In the third section we then outline some of the technical challenges that must be met to realize those advantages. The fourth section then describes advantages to be achieved by combining the different kinds of obliviousness provided by aspect-oriented and by service-oriented technologies. These advantages include not only increased malleability, but also a basis for describing statically enforced future processing commitments. These commitments can be used to merge process-flow and call/return paradigms and naturally express latent parallelism, to better exploit multi-core processors.

2. Underlying Issues

2.1 Malleability

Malleability is much like reusability, except that instead of characterizing how an artifact can be reused during the development of new artifacts, malleability characterizes how flexibly it can be used at deployment-time or run-time. For example, using Java's ability to describe the type of a parameter with an interface rather than with a class makes methods both more reusable and more malleable. But unlike Java, ADA and Modula-3 can identify a parameter's purpose by name. The order in which these names

* This work was made possible by a grant from the Science Foundation Ireland

are used by a method's caller can differ from the order in which the implementer listed them. This improves the malleability of both. But it does not improved their reusability because the order used by the selected implementation is evident at development time in any case. The information needed to reorder parameters can be provided without undue burden on the developer by referring methods and parameters defined in interfaces to a glossary that, like JavaDoc, provides extra-lingual information about meanings.

Generally speaking, malleability cannot be achieved by adding one or another language feature to address it, although features that increase the specification content of software over its algorithmic content add to software's malleability. It is instead easier to enumerate characteristics of software that inhibit malleability, and propose their removal or the substitution of equivalent characteristics in different form. We mentioned sensitivity to parameter order as one example, above, but there are many other ways that two implementations of the same function can differ. A trivial example is tolerance for name variations to be used for methods or types, which can be resolved through the use of the same glossary mentioned above. A more subtle inhibitor to malleability is the assignment of the implementation to a particular "target" object, which is otherwise just one among the several parameters. We have called the ability to ignore such differences "structural abstraction" [8] and defined a Java-compatible programming language called Continuum that permits the objects responsible for method implementations to be imagined differently by clients and services [9],[14].

2.2 Obliviousness

Obliviousness is an important way to achieve malleability. That is one of the reasons it is so important in separating concerns. Using structural abstraction, the service-oriented model for software provides a way to make clients oblivious to the issue of where a method is implemented within a service. But it does so by introducing a new structural dependence. By modeling services as objects, it replaces dependence on the assignment of methods to objects with dependence on the assignment of methods to services. As with traditional target-directed object calls, service requests and responses are *point-to-point*, forcing the client to rigidly reflect the realization of function by services.

On the other hand *obliviousness* [1] is one of the hallmarks of aspect-oriented technologies, which hide service attachments from clients. The concept of obliviousness recognizes that the flow of logic within software is not sensitive to independently described aspects that may each carry part of its state. Aspects can be used for attachment of systemic function like management of transactions, or for composition of component functionality like editing, display and validation of the elements of a development environment. No matter which, the fact that the combined aspects' code is oblivious to the manner of their combination is a major contributor to the software's malleability. However, today's exploitation of aspect-oriented concepts is only applied where clients do not control or direct the aspect code. It is not appropriate for modeling service attachment to clients because, after all, the conventional model of method call forces the client to know the method's implementer.

In [9], we observed that structural abstraction can be achieved by changing the nature of method call from point-to-point to broadcast. This is a deep change conventional object-oriented semantics in which method calls are always directed to a target object in a point-to-point fashion Rather than statically binding methods to classes, the face merely indicates that the methods' implementations have been demonstrated to be available. The same flexibility can be used to organize services in a manner making clients are oblivious to their structure.

2.3 Using Broadcast Method Call for Malleability

We will use a family of related examples shown in Figure 1 to illustrate several points related to use of structural abstraction and broadcast for method call. The problem being addressed is motivated by the commonly seen phenomenon of mobile phones' ability to optionally display a clock on the phone's window. Presuming a method "display" that a client can use to put the clock in the display – one can ask how to write the method call that invokes it. The lower part of Figure 1 shows such a client. To focus on structural abstraction, we ignore the grey-shaded material. The upper part of Figure 1 shows several possible server implementations. In (a), "display" is implemented in the window object, while in (b) it is implemented in the clock. In the interest of malleability, we wish the client to use either service supplier, where in this case we might presume that the "service" is local and in the clients classpath. The use of both interface and face declarations in the client is intended to highlight the fact that a legacy client may have expected the implementation to be in "Window". Treating interfaces as a "sugar" allows use of the legacy style.

The syntax in these examples is familiar in form, but has subtly different meanings from what may be expected. Full description of the type model supporting structural must be left elsewhere [3], as it would consume the space allotment for this paper. In brief, there is a classification hierarchy for objects in which a classifier can have more than one super-classifier. Classes, as distinct from classifiers, define the state and method implementations for objects, and are attached as leaves to the classification "tree". Classifiers

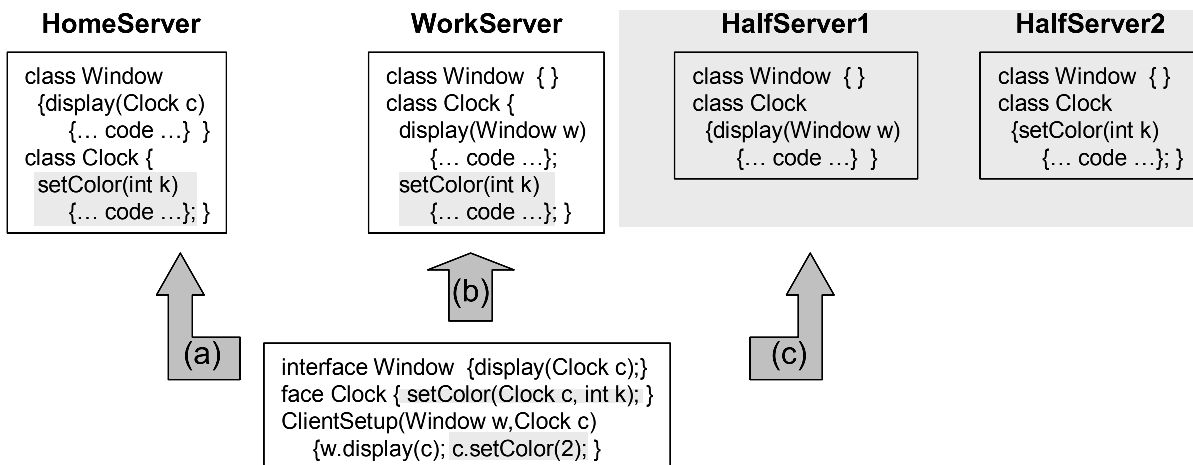

Figure 1 – Examples of Different Server Combinations for a Client

are not statically tied to sets of methods, which are called "faces". Instead, the declaration of a reference variable indicates a classifier, a face, and whether the reference value may be null. If the reference is not null, the methods in the face are assured to be accessible by the dispatcher. The set of assured methods can grow over time, and any one declaration need mention only a subset of the assured methods. The methods' implementations need not lie in the object referenced by the reference. They may be in any of the required parameters, or even statically available. The client need not know the service provider's class structure at all, which characteristic we call *structural* abstraction. We treat declaration of an interface as a syntactic shorthand for a face in which all of the methods have an additional parameter with the appropriate defaulted classifier. The net effect of this type model is that the client in Figure 1 is type-compatible with any of the servers.

Broadcasting method calls increases the malleability of software artifacts. It can do this because it replaces the concept of an interface that indicates methods available from an object with the concept of the *face* that indicates methods available somewhere in the environment. In the usual model, methods in an interface are available only if the associated reference is not null. As with conventional point-point interfaces, for broadcast the methods in a face are also only assured to be available if the associated reference is not null.

Interfaces are quite useful as vehicles for labeling the known contents of objects as they come from their implementer. But from the point-of-view of a function's consumer, what is of interest are the behaviors themselves, not which behavior is implemented by which object. So the "face" concept that replaces it identifies a set of methods on a variety of objects that must be available in the computing environment – in the "cloud", so to speak. A method call is not directed to an object, but is broadcast through the cloud to an implementation. The implementation has been proven, through the type system's interaction with the dispatcher, to be available somewhere. The difference is illustrated in the transition from Figure 1a to Figure 1b. When connected to the HomeServer (a), "display" has an implementation in the Window, as the client seems to expect. But the client can equally use the WorkServer (b). The client's use of a "target" in the call, as in the example's call to "w.display(c)" does not imply that the message is delivered to the target. In the example, since both window and clock are required (i.e. may not be null), it may lie in either. The actual target of the call is not defined by the language but by an extended dispatcher, the service-request broker, which finds the target's service as well as the class implementing the method, as described in the next section.

2.4 Using Broadcast Method Call for Obliviousness

Broadcasting method calls combines structural abstraction's obliviousness to object structure with aspect-oriented software's obliviousness to service structure. Doing so increases the malleability of software artifacts further above either structural abstraction or service-oriented approaches. The service request broker tracks the availability of services and routes called methods to the appropriate object in the appropriate service, freeing clients from the knowledge of the object or service that implements them. The grey-shaded material in Figure 1 focuses our attention on obliviousness to service structure. The services can be local to the client's classpath, or distributed elsewhere. The figure illustrates a composition of two services (c) that supplies the same needs as did (a) or (b) as single services. It shows a situation in which the displayed clock has settable state information to control its color, made available as a "setColor" method.

To completely separate client from the implementation's structure, we do not allow a client to name classes (implementations) at all. This carries the use of interfaces for characterizing object types to an extreme. Unless employing a factory pattern, the most common practice today is for a developer to instantiate a class whose characteristics are known to meet the functional needs, rather than to leave that selection to be made at run-time. This, again, implies knowledge of the class structure of the anticipated provider of services. To completely decouple the client from the service structure, we instead simply specify the classification (how the class must relate to its subclasses) and the face (which methods that are needed). In some cases, the provided set of methods may need to result from the composition of several available services, carried out behind-the-scenes from the client that creates the object.

In better separating client from object structure, the broadcast model's *face* also better separates clients from service structure. The fact that service boundaries are transparent allows the service model to be used at fine granularity. While we want to allow services to be distributed and mobile, for services to be composed locally, within a single process, and it is important that a two-level dispatch be avoided. The client does not target the method to a specific service provider, but allows the service-request broker to find and direct the call appropriately. If we include the grey-shaded material in examining Figure 1, we note that when supported by the composite server (HalfServer1 and HalfServer2), the client's apparent call to a the "display" method in Windows is actually implemented in HalfServer1's Clock, while the call to setColor, made with no specified target is implemented in HalfServer2's Clock.

3. Broadcast Service Model Challenges

We can foresee several challenges in trying to move from the current target-directed models for objects or services to the kind of broadcast service model that would provide the advantages described, including: compatibility, state maintenance, service visibility management, general service management, and commitment satisfaction tracking.

3.1 Compatibility

Any shift in programming paradigm will fail if it can not accommodate previously-written software. Even the successful shift from procedural to object-oriented was enabled by fact that C++ by definition included all of C, and the ongoing transition to service-oriented architectures is facilitated by treating services as objects within the object-oriented paradigm. It is therefore no accident that the broadcast service model for method call can include the conventional object-oriented model as a syntactic and semantic subset, though it is challenging to do so when eliminating the concept of target. As sketched in Section2.3, it preserves the concepts of a type hierarchy of classes (classifiers), of the association between classifiers and sets of methods, and of the fact that non-null references are required to assure the safety of method calls. But it permits parameters to be reordered and does not require the method implementation to reside in the target of the method call made by the client. Existing class implementations all function properly when interpreted as broadcast method calls rather than point-to-point calls.

3.2 State Maintenance

Many object-oriented systems maintain consistency using the simple premise that an object holds all its state and sees all of the method calls that make changes to the state. While simple, the premise is also frequently invalid in its over-simplistic view of the nature of state. For example, objects may indirectly access and return state maintained in other objects, in which case they will not see when the value changes by a call on the other object. In fact, as discussed in [4], the idea that an object has an objectively-definable state is itself limiting. Using an object's identity, other

objects may maintain additional data, whether in hash-tables or in aspects to which an application is oblivious.

Most object models rely on the idea that specifications about method dependence need not be included in a class specification because the object is guaranteed to have "seen" all methods called on it since its creation, in the order they are called. However, concurrent call and dynamic attachment of oblivious aspects violate this principle. On one hand, a dynamically attached aspect does not "see" methods called prior to its attachment. But on another hand, an object's state is not an opaque totality, but is the product of state contributions made by the independent aspects.

Because an object's state can be distributed among several services, a service that becomes newly available may not have an accurate picture of the object's state with respect to a client's prior calls within a transaction. To prevent inappropriate action, descriptions of the faces provided by services must include declarations that identify dependencies between method calls. Such dependencies are generically called *choreography*, and are recognized as important for service composition[16]. In Continuum[14], these constraints are expressed by indicating that a particular method is available only if all prior calls to specified other methods have been seen by the same service.

3.3 Service Management

Protocol constraints make it possible to determine that some services should not be visible to certain clients, but there can be other reasons, like cost, service-levels, or business arrangements that play as well. Current service-oriented systems generally manage visibility in a rather static fashion in which clients initiate a service-finding operation and then access the found service through a proxy. But this approach only works because the clients are dependent on the structure of the services, and would inhibit the kind of flexibility implied by the grey-shaded material in Figure 1. To free clients from this concern, the matching of clients to services is performed by the service broker, which can perform necessary bookkeeping with respect to the transaction, identified on each call.

But this does not serve the needs of dynamic, mobile environments well. If a service being used moves out of range, an alternative one visible for the client needs to be used instead. The service request broker is responsible for receiving communication from services joining the bus, and for managing their exit. In addition, the service broker must recognize that some services have a mutual awareness – they may be substitutable, as would be the local entry ports for commercial enterprises, or they may be incompatible or have other contractual relationships

In addition, the service request broker may perform *ad-hoc* composition of services needed to satisfy a client. If a client expects a face providing services for managing both air and hotel bookings and the available services provide one or the other, the broker can compose the services into a larger structure automatically, rather than requiring that the aggregated service be implemented particularly to perform both functions or coordinate both services.

While today's service-brokers could be imagined to provide an appropriate place for managing these functions, the fact that clients must specifically recognize distinct services as objects as discussed in Section 2.2 makes extending their capabilities cause changes to the clients. However, use of a broadcast service architecture allows the capabilities to be provided to clients transparently, without disrupting their operation.

3.4 Static Tracking of Commitment Satisfaction

There is a traditional gulf between object-oriented programming languages and work-flow architectures. In object-oriented languages, the client determines the target of a call, and waits for its completion. This is a powerful inhibitor to greater use of parallelism. In work-flow architectures on the other hand, the sender does not wait for its completion but target of the message is specified by the flow-designer. This provides many opportunities for parallelism, but the use of two architecturally disparate elements seems too cumbersome for use in algorithm description. This may be the reason the combination has not been adopted as a conventional programming language. The broadcast service model's use of a service request broker provides a novel way to integrate the concerns of programming language and work-flow architecture.

To exploit this capability, two additions are made to conventional programming language constructs: 1) the method call and message send constructs are unified, and 2) the concept of a statically declared "commitment to call" is introduced. A method may be declared to guarantee the future call of another method, as illustrated with the "`sends`" keyword in the `face` definition:

```
face X {void f1(A a, B b) sends f3(A a));}
```

This declaration defines a face, X, that declares support for a method f1 of two parameters. Method f1 commits to the eventual calling or sending of another method, f3, using the value provided by f1's first parameter.

Unlike conventional call's semantics, the static commitment to eventually call f3 need not be satisfied before the method carrying the declaration returns, but must be satisfied by the end of the outermost transaction in which the commitment is required. Thus, the commitment can be satisfied by the method itself during its execution or by the execution of methods it calls, or by a method to which a message is sent, perhaps much later than the client's completion.

A method's declaration may include a list of such static commitments. The method declaration can only be satisfied by an implementation that itself declares the satisfaction of the commitment. To enforce the behavior, the commitment must be satisfied on all paths from the entry of the method, either directly, or by *call* or by *send*.

Presuming this, another method, f2, which is also committed to send f3 can be implemented as:

```
void f2(A u) sends(f3(A u)) {
       // other computation
       send f1( u, new B() );      }
```

This implementation is valid because f2 sends f1 which is committed to send f3, thus satisfying f2's commitment. But if calls to f2 are made in a loop, only the "other computation" is serialized in the loop. The execution of the resulting f1's can all occur in parallel with the loop's execution.

As described in [6], dynamic failure to satisfy a commitment, whether by thrown exception or by reduction in resources can be handled locally, or it causes the transaction to abort.

Where appropriate, the use of static commitments also enables a *call* that would occur inside a loop to be transformed into a *send*. The service request broker can enable these activities to occur in parallel. Because the committed action is not guaranteed to take place immediately, the original caller can employ this mechanism only if further computation in that caller does not need to use the results. However, it is possible to define commitments in a way that enables subsequent gathering and processing of the results.

This alternative view of computation is made possible because unlike a conventional target-directed call, the use of a broadcast model allows the request broker to act in a store-and-forward capacity for parallel messaging in addition to the immediate-invocation-and-return capacity for conventional dispatchers.

4. Broadcast Service Model Advantages

The broadcast model enhances malleability by changing the programming language model to employ broadcast rather than point-to-point semantics for its call and to make clients oblivious to the structure of services. In doing so, it eliminates the need for a syntactically special target object on call. This is perhaps fortuitous, because instead of passing an implicit target, the language can instead reflect the concurrent structure of the software by passing a transactional context for bounding the action to be taken on failure.. The failure recovery points must be indicated directly within the code that engenders possible failures to permit us to write software with more latent concurrency than present.

4.1 Enhanced Malleability

The increased software malleability made possible by changing the programming model from the point-to-point model used by both classical object-oriented programming and popular service-oriented architectures to the broadcast model provides several malleability advantages:

Greater tolerance for different implementation structures. The un-shaded material in Figure 1 illustrates how a client needs no change to tolerate a different service provider that moves the display method either to different classes (since the implementing class need not be mentioned in the call) or to different services (since the service also need not be mentioned in the call).

Accommodation of dynamic service composition. The grey-shaded material in Figure 1 further illustrates how the client needs no change to tolerate a change to a different service provider structure altogether, since the service is not mentioned in the call. Since neither target objects and services are not mentioned in the call statements, combinations of services used to satisfy a client's needs can be fluidly composed by the service request broker.

Scalable component composition. Component structures like those employed in service-oriented software architectures suffer from severe performance problems when used at finer granularity in an attempt to obtain improved the software structure it offers [11].The use of transparent services, with a broadcast model of method call like that illustrated in Figure 1 enables the implementation to move the task of message and data format transformation out of the client and into the service request broker. This enables the associated overheads to be avoided when component structures are tightly bound within a process.

Avoidance of proxy management. In the usual division of concern between programming language and middleware, the programming language specifies the complete semantics of method call, including the rules for determining how to find the implementation corresponding to any particular method call. In architectures for distributed, autonomic, grid, and service-oriented systems, the linguistic specification is ultimately quite incorrect. The intervention of middleware takes the "call" out of the realm of language specification and makes non-linguistically specified choices. In fact, modern Object Request Brokers allow the client and service to be realized in different programming languages, making the specification of the dispatch process as a linguistic characteristic impossible. But the task of interfacing this flexibility with the language specification is forced upon the client in the form of "proxies" – local objects that intercept the linguistic specification and inject alternative mechanisms. Much greater flexibility can be derived if the client and the service provider left such intervention to the underlying implementation of the dispatch process – the request broker provided by middleware. Then the overheads associated with preparing for potential mismatches [11] could be omitted if the targets are near and have similar or identical signatures.

Reflecting middleware's flexibility in language's typing. Today's programming language specifications over-specify the interpretation of method-call, to the detriment of the software community in general. In systems that rely heavily on redirection via proxies, it would be more accurate for the programming language to specify only the semantics of the behavior occurring between entry and call, leaving the definition of a call's resolution to middleware. Language specification today is caught in a bind – to keep dispatch specification simple, the type systems generally require too much knowledge of the implementation structure to which a call is directed. Flexibility can be gained if they instead focused on accepting an indication from middleware about the safety of calling a method and propagating that information throughout the client. If the language specification simply carries forward a decision about the existence of implementation rather than trying to specify the matching rules, more flexible typing systems can be accommodated than those that require knowledge of the details of class implementations prior to run-time. The dispatch middleware then has flexibility in inserting conversions and rearrangements of the parameters, and even of employing different name-matching rules. What is required is a formal statement of the middleware's constraints, perhaps similar to the rule we propose: "the set of methods available to a client in a transaction is static or grows monotonically or the transaction fails."

Run-time selection of object classes. This same locus of intelligence applies when objects must be created. In traditional software, the implementing class of a new object is selected at the time a client is developed, generally after the developer inspects specifications for alternative implementations. With a service request broker, the client indicates what kind of object is needed and what methods must be made available for this kind of object. The "kind" is indicated by its classifier, with locally-defined meaning that allows individual services to describe subtyping relationships among different kinds of objects. Kinds of objects that support the same methods may still fall into different classifications because they attach different meanings to them. The focus is on characterizing the kind of object and the methods needed instead of on the implementing class. This allows the actual implementation class to be selected contextually, at run-time, by the service broker using new or local alternatives that may not have been known or available to the client's developer. It is also possible to augment the set of methods needed after objects have been created. This augmentation is known as *service-finding* and extends the idea of "down casting" in more familiar languages. When successful, the type system treats the knowledge that the methods are safe to call as if they were known to be available from creation.

Accommodation of service-substitution protocols. In mobile computing, and even in dynamically evolving system structures, it can be necessary to determine when one provider of services can be dynamically substituted for another. A contact for banking or travel information services may, for example, change in crossing regional boundaries. Or, some service providers may be more reliably reachable within one local region than another. The introduction of an intelligent service bus allows these issues to be addressed in a more organized and more easily maintained fashion than do proxies. Service providers interact with the service request broker when they are attached, and may provide information that helps determine their dynamic interchangeability.

4.2 Parallelism and Multi-core Support

4.2.1 Expressing Transactional Needs

If programming language design is to confront the issue of increased exploitation of parallel architectures, whether in distributed services or in multi-core machines, the ability to clearly delineate the transactional boundaries of failure of a concurrent element within must be provided. This is in addition to making provision for tracking the interferences of concurrent access to shared

data. While threads and transactions model the concurrency itself they have a natural intersection for the handling of errors. But programming languages generally have neither constructs to establish the boundaries of transactions within the execution nor a definition of their relationship. As with dispatch, the line between transaction model and transaction denotation needs to allow the detailed meaning and the implementation of transactions to be left to the middleware, but still permit the assignment of work to transactions to be expressed syntactically by the developer in a clear and direct manner. All execution takes place within a transaction. For convenience sake, the transaction in which an interpretation is taking place is best passed implicitly from caller to called method, as the thread is in familiar languages. But the language needs to allow for it to be explicitly specified on occasion. Explicit provision may be in the form of the creation of a new transaction or of the resumption of an existing one. While not proposing that the broadcast service model specify or restrict the transaction model unduly, it would be in line with many common specifications of transaction semantics for the transaction to be passed as an implicit parameter from client to service as method calls are made. Some provision must be made for changing the transactional context at the point of call. One possibility is to allow explicit change in the transaction context by using no-longer necessary syntactic position of the target object at the point of method call.

4.2.2 Static Enforcement of Task Commitments

One of the obstacles to greater exploitation of parallel structures is the fact that programming languages maintain commitment and failure response in a dynamic manner, using run-time interpretation to respond by waiting for the service to return. In addition to inhibiting parallelism, dynamic tracking is resource-intensive. Avoiding the need to hold resources has led to the exploitation of so-called "stateless services" in service-oriented architectures. But stateless services have no expression of flow dependencies between them. Since each service is finished before the next service acts, there is no way to express dependency on success or failure of later services, or to indicate back-out and recovery mechanisms. One common alternative is to combine them with separate process-flow specifications. While process-flow specification may be suitable for the niche in which service-oriented architectures operate, trying to use it to address the need for the widespread exploitation of parallelism called for by multi-core proponents is unlikely to succeed because it splits the specification into two language paradigms. At a coarse grain this may be acceptable, but at fine granularity it imposes too much intellectual and bookkeeping burden on the programmer. The service broker allows us to integrate process-flow more tightly into the usual programming language structures, in a way that allows a single program developer to exploit it easily.

5. Related Work

5.1 Broadcast Models

We are advocating the use of a broadcast model for method call to substantially improve the malleability of software. Broadcast models for processing have a long history of their own. One family of broadcast models center on a shared data-store. In that context, emphasis has been put on the use of *coordination languages*, like Linda [2]. Linda and subsequent tuple-space coordination systems provide primitives for controlling access by concurrent processes to a shaped data space of tuples. The point of intersection with message processing is that the database reading operations can wait for the appearance of a tuple matching an abstract template[13]. The effect of being able to wait for the appearance of a tuple matching an abstract template is much the same as the effect of a concurrent multiple dispatch, but the emphasis in Linda-based systems is in applications like data-mining, supported by a persistent tuple-store. We seek a replacement for the method call mechanism to remove layers of bookkeeping from clients and encourage malleability of software. The use of a separate coordination language or framework on top of the native language in a client scarcely makes it clearer or more malleable. In view of the performance overhead associated with Linda's point-of-view[13], other broadcast systems focus more on the delivery of ephemeral messages rather than a replacement for method call.

Non-storage-based systems, often called *message brokers*, play an important role in commercial systems, supported by products like IBM WebSphere MQ[17]. The primary advantage of such publish/subscribe systems is that the clients and servers need not be modified when message routing specifications change. Message frameworks like Java's JMS[15] have also been specified, but generally require the client to make static advance decisions about routing by specifying a class of object, like Topic, that manages where messages are sent. With these systems, the client typically uses a cumbersome framework that often involves data format conversions. This interferes with the transparency and malleability which is our goal. Academic interest in message brokers is thin, except when viewed as multi-methods.

5.2 Multi-methods

Methods that may be dispatched on the basis of the types of more than one argument are generally called multi-methods and are an area of significant and long interest. Although it is an object-oriented language, the invocation construct provided by CLOS [7] provides for structural abstraction to shield the client from the structure of choice-making in the implementation. But CLOS does not provide static typing, and its use was limited to the LISP community for many years. An excellent recap of work on multi-methods is given in [10]. It is important to note that most of this work is directed at achieving multiple-dispatch in languages that permit declarative typing, and not at hiding the dispatch criteria and implementation structure from clients, and they generally sacrifice either conditional safety or structural abstraction to do so.

Programming language research generally exploits multi-methods to specialize the behavior of methods for various argument types. This has led to thorough investigation of issues of ambiguity. In the absence of a modularizing structure larger than classes (like OSGI's "bundles"), restrictions that reduce the malleability of software have been used to introduce *modular* type-checking[10]. Pursuing malleability, we have exploited the concept of a language-defined service to bound the scope of possible ambiguity in addition to providing a separate container for service state.

5.3 Aspects

In addition to providing a malleable component construct, services can serve as aspects as well. Symmetric approaches to aspect-oriented software separate the materialization of the aspects, which contain state and method definitions that extend the semantics of objects from the expression of the pointcut specifications that indicate when they should be employed. FuseJ[12] provides a unified aspect/component model with these capabilities. With symmetric aspects, the pointcut or aspect interaction language is separated from the programming language and, in our case, expressed when services are introduced to the service request broker.

6. Summary

To improve the malleability of software, we have employed a model of method call in which a method's caller can safely declare and call methods without knowledge of which object(s) or service(s) implement the method. We exploit component model in which *services* form coherent collections of classes and manage

their supporting state. Method call is treated as a broadcast in which a call can be dispatched to multiple services through the operation of a service-request broker. Services can be composed locally or they can be mobile or distributed. The structure of the services used by a client is fluid and transparent, and the overhead of conversion or marshalling takes place in the broker and is avoided when components are locally supplied. The state associated with an object can be distributed across several services, much as it can be distributed across several aspects, enabling the services to be used as symmetric aspects. In addition to increasing the malleability of software, the service-broker construct permits the integration of the concept of transaction into method call semantics. Expressing transaction boundaries in the programming language permits an extension of the concept of responsibilities for future action. It is possible to use the static type system of a programming language to enforce the future execution of a method after the return from a method which commits to that future execution. This facility can be used to provide greater parallelism when such methods are called in loops.

Acknowledgements

I would like to thank the reviewers and especially Eric Eide for suggestions that have substantially improved this presentation.

References

[1] Filman, R.E. and D.P. Friedman: Aspect-Oriented Programming is Quantification and Obliviousness. *In: Position paper for the Advanced Separation of Concerns Workshop at the Conference on Object-Oriented Programming Systems, Languages, and Applications*, Minneapolis, MN, October 2000

[2] Gelerntner, D., Carriero, N, Coordination Languages and their Sgnificance, *Communications of the ACM*, *35,2*, (February, 1992), pp. 97-107

[3] Harrison, W., Lievens, D., Walsh, T.., Achieving Recombinance to Improve Modularity. Software Structures Group Report 102, October, 2006, available from https://www.cs.tcd.ie/research_groups/ssg

[4] Harrison, W. and Ossher, H., Subject-Oriented Programming - A Critique of Pure Objects, *In Proceedings of 1993 Conference on Object-Oriented Programming Systems, Languages, and Applications*, September 1993

[5] Harrison, W., Ossher, H., and Tarr, P., Software Engineering Tools and Environments: A Roadmap, in *Future of Software Engineering*, Anthony Finkelstein (Ed.), ACM Press, June 2000

[6] Harrison, W.. De-constructing and Re-constructing Aspect-Orientation, *In Proceedings of the Seventh Annual Workshop on Foundations of Aspect Languages*, Brussels, Belgium, 1 April, 2008, edited by Gary T. Leavens , ACM Digital Library, 2008, pp. 43-50

[7] Keene S., *Object-Oriented Programming in Common Lisp,* Addison-Wesley, 1989

[8] Lievens, D., Harrison, W.. Symmetric encapsulated multi-methods to abstract over application structure, *In Proceedings of the 24th Annual ACM Symposium on Applied Computing, Symposium on Applied Computing*, Honolulu, HI, March 8-12, 2009, ACM, 2009, pp. 1873 - 1880

[9] Lievens, D., Walsh, T., Dahlem,, D. Harrison, W.. Promoting Evolution Through Abstraction Over Implementation Structure, *In Proceedings Companion of the 31st International Conference on Software Engineering*, Vancouver, Canada, May 16-19, 2009.

[10] Millstein, T., and Chambers, C. Modular Statically Typed Multimethods. in *Proceedings of the 13th European Conference on Object-Oriented Programming (ECOOP 99)*, Lisbon, Portugal, June 14-18, 1999

[11] Mitchell, N., Sevitsky, G., and Srinivasan, H., Modeling Runtime Behavior in Framework-Based Applications, in *Proceedings of the 20th European Conference on Object-Oriented Programming (ECOOP 06)*, Nantes, France

[12] Suvee, D., De Fraine,B., and Vanderperren, W., A Symmetric and Unified Approach Towards Combining Aspect-Oriented and Component-Based Software Development*, in Component-Based Software Engineering*, LNCS 4063, Springer, Berlin / Heidelberg, 2006

[13] Wells, G., Coordination Languages: Back to the Future with Linda, Proceedings of the Second International Workshop on Coordination and Adaption Techniques for Software Entities (WCAT05), pp. 87-98, 2005.

[14] Continuum Language Specification, available from https://www.cs.tcd.ie/research_groups/ssg

[15] Sun Java Message Service (JMS), http://java.sun.com/products/jms/, retrieved 24 Jan 2010

[16] Web Services Choreography Description Language, http://www.w3.org/TR/2005/CR-ws-cdl-10-20051109/, retrieved 24 Jan 2010

[17] WebSphere Message Broker Technical Overview, http://publib.boulder.ibm.com/infocenter/wmbhelp/v6r1m0/topic/com.ibm.etools.mft.doc/ab20551_.htm, retrieved 24 Jan 2010.