Generalizing Dispatching in a Distributed Object System

Farshad Nayeri, Ben Hurwitz, and Frank Manola

GTE Laboratories Incorporated
Waltham, Massachusetts 02254 USA

Abstract. Today's distributed computing environment presents a jungle of systems that use different object models, programming languages, and paradigms. Taking maximum advantage of these diverse resources requires that they be able to interoperate. We report on a series of experiments in a distributed object system that show how a flexible notion of dispatching can be used to integrate objects belonging to different models, systems, and paradigms.

Introduction

Today, many different object models are employed in various programming language implementations, database management systems, expert systems, and operating systems services. This proliferation of object models hinders reuse because code written in one model typically cannot interoperate with code written in another. While global interoperability could be achieved by the adoption of a particular object model, it is likely that no one model will achieve universal acceptance. Hence, the best achievable interoperability solution may be to create mappings between the features of the various models. Achieving such inter-model mappings is complicated by the distinct features that distinguish individual systems, including:

- mechanisms for dispatching, such as classes or generic functions
- computing paradigms, such as imperative, rule-based, or functional
- object composition features based on inheritance or delegation
- synchronous and asynchronous communication based on messages or events.

In this paper, we focus on the process of *dispatching* and its variations in different object systems. Through a number of feasibility experiments, we show how we can integrate a diverse set of object models and paradigms by taking advantage of a flexible notion of dispatching in the context of our prototype distributed object manager, called DOM.

Organization of this paper. Section 1 explains why it is important to focus on dispatching in different object systems, briefly relating our work with the previous work in this area. Section 2 describes flexible dispatching and its implementation in DOM. Section 3 reports on a series of experiments to integrate several object models and systems using flexible dispatching. Section 4 surveys related work. Section 5 reflects on the lessons learned and some of the open issues. Section 6 concludes by summarizing the paper.

1 Motivation

Inherent in every object model is the mapping of invocations on objects to their behavior. *Dispatching* is the process of finding the appropriate behavior for a particular invocation. Most object systems provide a *single* and most often *fixed* form of dispatching that is *built into* the implementation of their object model. For example, the Smalltalk [1] system performs the following steps upon invocation:

```
find the receiver of the message
search through the class of the receiver and its superclasses
  until a class is found that contains a method with the same
    name as the message selector
if such a class is found then
  apply the method to the receiver and the other arguments
else signal messageNotUnderstood
```

Different object models employ different dispatching mechanisms in order to provide support for their flavor of computation, or to address implementation goals. In some models, such as C++ [2], dispatching takes place implicitly through subclassing, while in other models, such as Self [3] dispatching plays a more explicit role through delegation. CLOS [4] allows for some variations on the basic theme of dispatching, while keeping the overall process fixed.

DOM is a distributed object system intended to support interoperability among heterogeneous components. Because our main goal is to integrate objects from different object models and systems, we do *not* assume a single, fixed form of dispatching. Instead, DOM allows different objects to employ different dispatching paradigms. This is achieved by requiring object implementations to specify explicitly how dispatching should take place. Hence we allow for objects belonging to different dispatching paradigms to coexist within the same system.

After collecting the common kinds of useful dispatching patterns, we can define higher-level abstractions that seem useful and safe in practice. In this sense, our explicit notion of dispatching can be considered "the 'goto' of object run-times": it serves as a crude but powerful method of defining a portion of the behavioral aspects of an object model. We believe that, just as was the case with the development of programming languages, gaining familiarity with explicit lower-level constructs such as dispatching will help us define better higher-level abstractions needed to integrate future distributed object systems.

2 Dispatching in DOM

From the perspectives of the DOM *user* or the *client*, the most fundamental abstraction in the object model is the *invocation*. Every request for object functionality is an invocation. The body of an invocation is a grouping of objects $(o_1\ o_2\ \dots\ o_n)$.

When DOM receives an invocation, it picks the first argument in the invocation, called the *receiver*, and forwards the rest of the arguments to the receiver. The real responsibility of finding and executing the code implementing the invocation semantics rests with the receiver.

From the perspective of the *implementor* of an object, the most fundamental abstraction in DOM is the object's *dispatcher.* A dispatcher is a piece of code; when an invocation is forwarded to an object, the object's dispatcher is executed. All objects within DOM – including primitives such as integers or strings, add-on abstractions such as object ids and classes, and "foreign" or legacy systems – provide their functionality to the system by specifying their dispatchers.

The role of the system is to translate incoming invocations into calls to dispatchers of receiver objects; the dispatchers in turn determine and execute the appropriate pieces of code in response to invocations. Since the system makes no assumptions about the representation or behavior of an object, the object can be implemented in practically any language or any system, as long as it provides a dispatcher to act as an interface between the system and the object's functionality.

Bootstrapping core objects. Starting at the lowest level of the system, the objects representing basic data types – such as integers, strings, symbols, and vectors – are called the *core objects* or *core primitives.* DOM uses core objects to implement its basic functionality. In DOM, core objects do not have a special status; they are implemented like ordinary objects by designating dispatchers. As a trivial example of using a dispatcher to define object functionality in DOM, we present the Modula-3 code that implements integers in DOM:[1]

```
TYPE Integer = Obj.T OBJECT
      value: INTEGER;
    OVERRIDES
      dispatcher := IntegerDispatch;
    END;

PROCEDURE IntegerDispatch (self: Integer; args: Args.T): Obj.T
                RAISES {Obj.Exception} =
  VAR
      selector := Args.GetSelector (args);
  BEGIN
    IF (Text.Equal (selector, "printString")) THEN
      (* return a a printed representation. *)
      Args.CheckNumberOfArguments (args, 1);
      RETURN MakeString (Fmt.Int (GetInteger(self)));
    ELSIF Text.Equal (selector, "add") THEN
      (* add the argument to self, return the result. *)
      Args.CheckNumberOfArguments (args, 2);
```

[1] Modula-3 [5] is the underlying implementation language for most DOM primitives.

```
        RETURN MakeInteger(GetInteger (self) +
                               GetInteger (Args.Element (args, 1)));
      ENDIF;
      (* could not find the requested selector. *)
      RAISE Obj.Exception (Exception.badFunction);
    END IntegerDispatch;
```

By creating a Modula-3 type Integer, we have provided for all the integer objects within the system to share the same dispatcher. The dispatcher for integer objects responds to two messages: printString with no arguments, and add with one argument. Otherwise, an exception is raised.[2] The basis of *flexible dispatching* in DOM is the ability to create such customized dispatchers for objects using different paradigms.

From invocations to dispatching. To make invocations in DOM, most clients call the Modula-3 procedure Obj.Invoke. This procedure provides a uniform method for invoking operations on all DOM objects. For example, the following Modula-3 fragment defines two Integer objects using the Modula-3 NEW primitive, and invokes an add operation on them:

```
IMPORT Obj;
VAR
  a := NEW (Integer, value := 5);
  b := NEW (Integer, value := 6);
  c := Obj.Invoke (a, "add", b);
...
```

Within Obj.Invoke's body lies the assumption that the *first argument of an invocation is the receiver for a message.* Upon invocation Obj.Invoke calls the receiver's dispatcher, including the other arguments in the invocation. Obj.Invoke hides the notion of dispatching from clients; all the clients know is that they are invoking an operation on an object.

Obj.Invoke assumes that its second argument is the name of the invoked operation. Using Obj.Invoke denotes a message-passing style of invocation, which we most often use. DOM provides a more general invocation procedure Obj.InvokeN, which does not assume the message-passing style.

Scripting Language. Some examples in this paper are written in the DOM scripting language. The scripting language is not an essential part of DOM; it is merely syntactic sugar for defining and using objects more easily. It is not a full-fledged programming language, but the bare minimum for our experiments.

We followed the spirit of Scheme [6] by providing a simple, lexically-scoped interpreted language. Special forms such as DEFINE and IF provide their own evaluation rules, and are in upper case by convention. All other expressions

2. Efficiency of core objects is not a direct goal here, since we envisage computation-intensive tasks taking place on local systems, and that the core objects be used only to communicate between different systems.

are either variable references, literal objects, or invocations which take the form $(o_1\ o_2\ ...\ o_n)$. To process an invocation, the interpreter for the scripting language evaluates all objects in an invocation and calls Obj.InvokeN with the invocation as an argument. Obj.InvokeN will then call the dispatcher of the first object in the invocation. For message-passing invocations, the second argument – the message name – is a symbol, as is the case for (aRectangle 'draw). For functional invocations, the second argument of the invocation is not restricted to be a symbol. For example, the object aRectangle is an argument of an invocation on the function draw in (draw aRectangle). The following is a DOM script that parallels the above Modula-3 fragment:

```
(DEFINE a 5)
(DEFINE b 6)
(DEFINE c (a 'add b))
```

The scripting language itself was implemented using DOM primitives. One of the interesting applications of flexible dispatching is the implementation of *closures* in our scripting language.[3] Closure objects in DOM contain a parse-tree representing the *body* of the closure, and a symbol dictionary representing the *environment* where the closure was defined:

```
TYPE
  LambdaClosure = Obj.T OBJECT
    body: Obj.T;
    environment: Obj.T;
  OVERRIDES
    dispatcher := LambdaDispatcher;
  END
```

Upon invocation, the dispatcher of a closure binds the invocation arguments to the formal parameters of the closure and evaluates its parse-tree in its environment. This implementation is straightforward since the notion of a dispatcher in DOM is similar to that of a closure in Scheme. The difference is that by relaxing the implementation requirements on objects with dispatchers, we can allow them to be implemented in a variety of languages and systems.

Comparing the LambdaClosure definition to the Integer definition given earlier shows how DOM objects differ from each other: in their state, and in the procedure acting as their dispatcher.

3. A *closure* is a "free function" that gets created when a LAMBDA special form is evaluated. For example, (LAMBDA (x) x) evaluates to the identity closure.

3 Experiments with Dispatching

In this section, we report on some of the experiments that we performed in our study of dispatching in object systems. The two major goals for our experiments were:

- *feasibility* – show that employing a simple but customizable notion of dispatching allows us to integrate diverse object models
- *observing commonality* – observe the common patterns of dispatching in different systems.

In our experiments, we attempted to examine a wide range of models, systems, and paradigms, including:

- class-based and generic function-based object models
- client-server and distributed object concepts
- object-oriented databases
- rule-based systems.

Using DOM, we have integrated code written in a wide variety of systems and models, including: Modula-3, C, C++, Macintosh Common Lisp, CLIPS rule-based system, Sybase relational database, and Ontos object-oriented database system. As our experiments show, the flexible dispatching design of DOM made it easier to integrate diverse systems and models.

3.1 Class-based dispatching

In a classical object model, invocations are based on a *message-passing* paradigm: an invocation is interpreted as a message sent to a particular object in the invocation, called the *receiver*. The outcome of dispatching for a particular form is *solely determined by the receiver argument in an invocation*. Thus, to handle a new kind of message, the programmer needs to add a new handler in the code for the anticipated receiver of the message.

Many classical object systems use the notion of a *class* as their central abstraction and grouping facility. Such systems are usually called *class-based*. Classes typically play one or more of the following roles in a class-based system:

- *implementation sharing:* a central place for a group of objects to share behavior
- *interface sharing:* a central place for a group of objects to share external interfaces
- *factory:* an object that can create new objects that share behavior or interface.

Traditional object systems such as Smalltalk make fundamental assumptions about the role of classes as built-in parts of the overall model of the system. Classes in such a system are often treated specially. More recent object model designs for systems such as Emerald [7] and Self [3] do not include a built-in notion of class. Instead, they employ notions such as *delegation* to provide capabilities similar to the *inheritance* capabilities provided by most class-based systems.

In this experiment, we explore how we can use DOM's flexible dispatching to support the implementation and interface sharing roles of classes in an object system.[4] To do this, we first examine the typical characteristics of dispatching in a class-based system:

- each object has a class
- classes have superclasses; the set of all superclass relationships is the *class hierarchy*
- when a message is sent to an object, the system searches the class hierarchy in some predefined order to find a piece of code (a *method*) whose name matches the selector for the invoked message.

This class-based dispatching process is usually fixed as part of the object model of class-based systems. In addition to the characteristics listed above, many object systems make other assumptions about dispatching. For example, in Smalltalk, objects cannot include individual behavior outside the context of their class.

In DOM, since we do not necessarily know the representation or behavior of objects within the system, we cannot make such unilateral assumptions. Instead, we allow for the class notion to be an add-on service applied only to objects participating in a class-based model. Other objects do not have to comply with restrictions of the class-based paradigm. (In particular, DOM can support multiple class-based paradigms.)

Class-based dispatching in DOM. The essence of our implementation of classes is similar to that of the traditional class-based model. Objects adhering to a class-based paradigm contain a reference to their class, and provide a "custom" dispatcher that implements class-based dispatching:

```
TYPE ObjWithClasses = Obj.T OBJECT
      class: Obj.T;
    OVERRIDES
      dispatcher := ObjWithClassesDispatch;
END;
```

The dispatcher for class-based objects searches up the superclass hierarchy to find a method whose name matches the selector for the invocation:

```
PROCEDURE ObjWithClassesDispatch (self: Obj.T; arguments: Args.T):
Obj.T RAISES {Obj.Exception} =

VAR
    selector := Args.GetSelector (arguments);
    class := self.class;
    op: Procedure := NIL;

BEGIN
```

[4] We omit the discussion of the factory role of classes in this paper.

```
        (* Search the inheritance hierarchy to find the right class. *)
        WHILE class # RootClass AND op = NIL DO
            op := Lookup (selector, class);
            class := Obj.Invoke (class, "super");
        END;

        (* If a "method" is found, apply it; otherwise raise an error. *)
        IF op = NIL THEN RaiseException ("method not found"); END;
        RETURN ApplyPieceOfCode (op, self, arguments);

    END ObjWithClassesDispatch;
```

A different class-based paradigm (for example, one that uses multiple-inheritance) can be implemented by defining a new dispatcher, and possibly more state for the participating objects. Also, as in Self [3], we can avoid the meta-regress problems of systems such as Smalltalk, by implementing class objects themselves using a different dispatcher from ObjWithClassesDispatch.

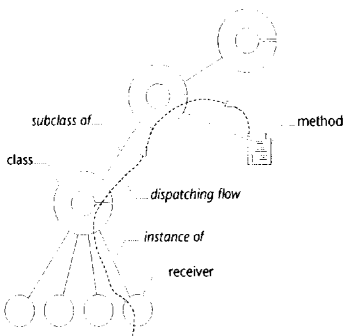

Figure 1. Dispatching in a class-based system: the system searches the class hierarchy beginning with the receiver's class to find the method to execute.

3.2 Generic function dispatching

Sometimes it is useful to consider parts of an invocation other than the receiver in the dispatching process. Chambers [8] provides some examples:

- (aShape 'draw aDevice) where aShape is a graphical shape and aDevice is a graphical device, such as a laser printer or an X Window. In this case, the piece of code that needs to be run depends not only on the kind of shape we are drawing, but also on the kind of device we are drawing onto.
- algebraic operations such as arithmetic (add or multiply) or ordering (=, <, >).

In the above cases, the decision of which piece of code to run is based *not only on the receiver, but also on other arguments of the invocation*. In this experiment, we describe how we can modify the dispatching process to involve all the arguments in an invocation without changing DOM's basic model of computation.

Consider coding the Shape object in the first example above in the classical model. One solution is to do a case analysis on the second argument of the invocation (aDevice) in the body of the receiver (aShape), but this solution produces code that is error-prone and hard to maintain. To avoid this case analysis, we can apply a technique called *double dispatching* [9]. In a classical object model, the only significant argument in the dispatching of a message is the receiver; to make a different argument significant, we must make it into a receiver. We do this by switching the order of arguments in the code for the draw method of aShape. This introduces a case analysis problem in the body of the Device objects. To avoid this second case analysis, we tag the draw message (sent to the Device object) by the different kinds of Shape objects (to create, for example, drawRectangle). By tagging messages, we are making the *case analysis implicit within the dispatching process.*

The main disadvantage of double dispatching is that it relies on the discipline of the programmer in making sure that the messages are forwarded correctly. Also, in the case of *multiple dispatching,* where we may want to dispatch on more than one argument, we will have to write code to repeat the same kind of technique for each significant argument.

Generalized object systems such as the Common Lisp Object System (CLOS) [4] support groupings of pieces of code, called *generic functions,* that differ from those used in classical object models. A generic function is a grouping of methods that implement the same functionality over some set of classes in the system. A CLOS programmer considers "draw" to be a generic function, since all draw methods implement variations of the same functionality. In CLOS, definitions are centered around the generic function itself. CLOS does not support a notion of a receiver; *all* arguments of a particular operation are considered in dispatching. Methods specialize on one or more arguments of a generic function, by providing a piece of code to be run when the arguments of a generic function match the types specified in the method declarations. For example, CLOS code to implement the draw generic function to support dispatching based on Device objects looks like:

```
(defgeneric draw (aShape aDevice))
(defmethod draw ((aShape Rectangle) (aDevice X-window)) ...)
```

Generic functions provide a convenient grouping when the dispatching decision is based on more than one argument. In the generic function approach, there is no need for the programmer to redirect the dispatching. On the other hand, because of its generalization, the system's dispatching process is more complicated.

Generic functions in DOM. We aimed to capture the essence of the generic function concept, while keeping within the classical model of invocation. We started by observing that the computational model of CLOS is centered around the concept of a generic function, or an *operation,* being applied over a number of objects.

To implement generic functions in DOM, we aimed to maintain the operation grouping explicitly by creating a *generic function object* which provides for registering pieces of code that together implement a particular generic operation. Here, DOM's flexible dispatching plays a key role: upon invocation, the dispatcher of a generic function object delegates the work to the appropriate piece of code depending on the parameters of the invocation.

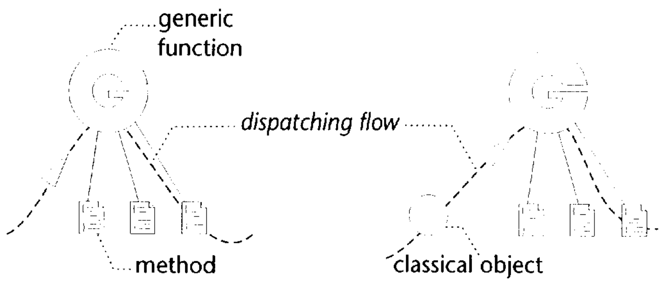

Figure 2. Generic functions in DOM (left) and how they can be integrated with classical objects (right).

Merging the classical model with the functional model. We can provide generic function capabilities for ordinary objects in DOM which adhere to the classical model. For example, suppose we were to create shape and device objects that support a draw operation so that (aShape 'draw aDevice) would mean to call the draw operation of a shape aShape on a device aDevice. To support this form of invocation, all we have to do is to ensure that upon invocation, the dispatcher of aShape calls the generic function draw while passing arguments (including the receiver aShape, but excluding the selector 'draw). In this sense, aShape is a classical object (i.e., it has its own dispatcher), but its dispatcher is not classical (i.e., it supports the generic object model).

Implementation details. Code for defining the generic function draw in our scripting language looks like:

```
(DEFINE draw
  (GENERIC-FUNCTION (shape device))

(ADD-METHOD draw (shape device)
  (AND (is-rectangle shape) (is-printer device))
    code for drawing a rectangle on a printer...)

(ADD-METHOD draw (shape device)
  (AND (is-circle shape) (is-window device))
    code for drawing a circle on a window...)
```

Like the CLOS Metaobject Protocol (MOP) [10], we have added a layer of syntactic sugar (special forms GENERIC-FUNCTION and ADD-METHOD) to ease programming. GENERIC-FUNCTION simply creates a new generic function object. ADD-METHOD turns its condition clause into a syntactic closure to be evaluated when the generic function is invoked. Upon invocation of a method, if the condition closure evaluates to true, the body of the method is executed. Since closures are just like ordinary objects in DOM, we can define a generic function from outside the scripting language by replacing the condition closure with an ordinary object invocation. Like Dylan [11] this design allows us to run methods as "type-checked" closures outside the context of a generic function.

Unlike CLOS, we do not assume the existence of classes. To augment generic functions to include a class notion similar to that of CLOS, we must extend the class-based dispatching to provide:

- a class membership test for instances
- a way of checking whether a class implements a particular invocation

The generic function dispatcher can use the information provided by the class system to find the appropriate method, similar to CLOS MOP. We can change the generic function dispatcher to provide for CLOS-style "before" or "after" methods and "eql specializers".

Like Dylan, we treat generic functions and methods as first-class citizens of the object system, providing the usual benefit of uniform access. One consequence of this choice is that DOM may allow two different generic functions to have the same name in different contexts, for example, (draw aShape aDevice) vs. (draw aGun). CLOS distinguishes generic functions *by name*, thus relying on namespace management to address possible conflicts.

3.3 Dispatching to Distributed Objects

The notion of a network message between components in a distributed system can be realized as an invocation in an object model [12, 13]. DOM's basic model of computation does not include a built-in notion of remote invocations. By utilizing flexible dispatching, we can allow for the remote invocation without changing DOM's basic model of computation.

In this section, we describe how we used flexible dispatching to build the two key abstractions of our distributed object implementation. A key characteristic of our design is that it does not enforce a canonical representation for object references. Instead, each system using DOM is free to define its own representation for object references. We also address the issues of concurrency and describe how network dispatching can be optimized.

Network objects: client-server computing in DOM. The client-server model of distributed computing can be mapped to DOM's basic model of computation in the following way: the *client* (invoker) invokes an operation on the *remote server* (receiver). To extend the local notion of invocation to a distributed one, we must:

- modify the dispatching to allow for calls to remote servers over the network (i.e., create a new dispatcher that allows for network invocations)
- provide an object to represent the server in the client's system. The client can make invocations on this object.

We introduce the abstraction of a *network object* to be a local representative for a remote server. (Others have called network objects "surrogates" [13] or "proxies"[14].) Upon invocation, the dispatcher for network objects performs the following steps:

- open a connection to the server it represents
- convert the invocation arguments to a network-neutral format
- send the contents of the message over to the server it represents; wait for server to respond
- convert the server's response from the network-neutral format to the local format
- close the connection
- return the reply of the server as its return value.

Somewhere on the local system, we must leave enough information so that the network object can forward invocations to the network server it represents. In DOM, the network object itself contains the information it requires for making network connections to the server it represents. In this sense, a network object encapsulates the local portion of the dispatching process from the client, delegating the rest of dispatching to the remote server. Under TCP/IP, the representation for network objects needs to contain a host name and a port number:

```
TYPE NetObj = Obj.T OBJECT
  hostname: TEXT;
  portnum: CARDINAL;
OVERRIDES
  dispatcher := NetObjDispatcher;
END
```

Network objects not only allow for the extension of DOM's basic computation model to include the notion of distribution, they also allow objects to be implemented in virtually any language, system or even networking protocol, while keeping the user's abstract model the same. For example, using network objects we can invoke operations remotely on a name server or on a legacy system.

Exporting object references. There are times when a server needs to manage a large number of individual objects. In these cases, it is often convenient for a server to export references to objects residing on the server to the outside world. For example:

- a file server may want to pass a handle to a file

- an object-oriented database may want to hand out references to objects in the database
- a process may want to send references to its DOM objects to other processes in the system.

We call such a reference an *object id*, or *oid* for short, and a server capable of exporting references a *component*. Using the oid for an object, the client can directly invoke messages on the remote object without having to worry about the component to which the oid belongs. Since an oid must reference an individual object within a server on the network, it must contain, at least:

- *a network address*: the network location of the component where the referenced object belongs
- *an object index*: an identifier for the object within the remote component.

Network addresses can be represented by network objects. Our design does not make any assumptions about the representation of the object index, except to preserve it across invocations. The interpretation of the index can be left as the responsibility of the remote component. In DOM, oids are first-class objects. Since DOM does not enforce a canonical representation for oids, each component is free to define its own representation for oids. The representation of oids in Modula-3 looks like:

```
TYPE Oid = Obj.T OBJECT
  component: NetObj;
  index: INTEGER;[5]
OVERRIDES
  dispatcher := OidDispatcher;
END;
```

Supporting invocations. We also have to make sure that local invocations on the oid trigger invocations on the object within the remote component which the oid represents. First, we need a way of instructing components to make an invocation on an object residing within that component. We do this by enforcing a uniform protocol across components for network invocations. All components are required to handle the network message invoke that takes a vector of objects representing arguments in an invocation, for example:

```
(file-server 'invoke (VECTOR "/proj/foo" 'append "/proj/bar"))
```

Sending an invoke message to a component triggers an invocation on that component.

5. For simplicity, in the implementation described here, we assume that the object index is always an integer.

We program the dispatcher for oids to:

- package the arguments into a vector
- execute an "invoke" operation (with the argument vector) on the network object representing the remote component
- return the result from the invocation made on the network object.

For example, the invocation (an-oid 'add 42) is translated into:

```
(a-server 'invoke (VECTOR an-oid 'add 42))
```

where an-oid is an oid referencing an object exported by a-server.

Oids as logical references. We can generalize the notion of an oid to include a logical identifier for components. This gives components freedom to relocate to different port numbers or machines on the network without affecting their clients. To implement this, we replace the network address in an oid with a *component identifier.* The mapping from component identifiers to network objects is maintained by a simple network service called the *location broker* or the *locator* for short. We change the dispatcher for oids to ask the locator for the network object representing the remote component. The dispatcher then uses the obtained network object to make the remote invocation. Hence, components can migrate readily from one machine to another without affecting their clients. In this role, DOM resembles an Object Request Broker (ORB) [15].[6]

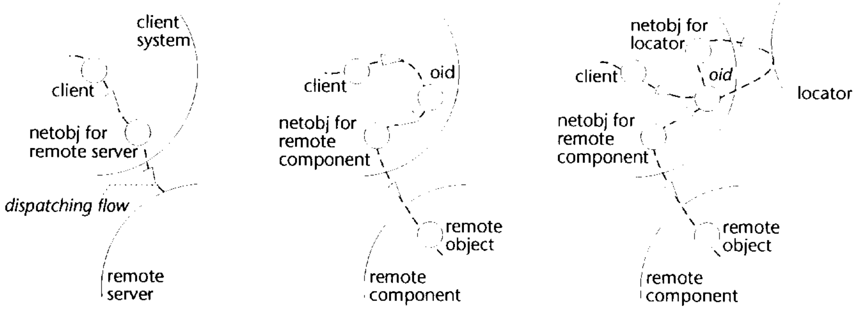

Figure 3. Three levels of network interaction: client/server interaction (left), using oids to encapsulate servers (middle), and using the locator to get rid of location dependencies (right).

Concurrency and DOM's basic model of computation. A component may need to respond to network invocations at any time. Often, when a network invocation arrives, a component may be servicing an ongoing task, for example, an invocation from a local script interpreter. Using Modula-3 *threads*, we can allow for multiple concurrent invocations on objects in DOM. To service

[6] To make the design scalable, we must allow for the location service to be replicated. We do not address this issue here.

network requests, each component spawns a thread upon start-up which waits for network invocations after registering the component's network address with the locator. The component then goes on to perform its local tasks as if it did not have to deal with network interactions. To avoid deadlock, we spawn a new thread for servicing each network invocation. We have protected the critical sections in DOM code. Protection of the critical sections within the code for the object itself is left to the programmer.

In the absence of a multiple-threaded environment, it is possible to achieve similar behavior by using an event-driven invocation mechanism or by supporting asynchronous message-passing [16]. We chose this synchronous model of invocation because it allowed us to keep DOM's basic computation model simple [12].

Optimization. In our first implementation, we sacrificed efficiency for clarity and simplicity of code. Because of this, we held off on implementing all optimization strategies. Once the basic model is implemented, we can apply a range of optimizations. For example, *memoizing* or *caching* is an optimization technique that is common in implementations of both Metaobject Protocols [10] and distributed systems [12]. In our implementation, there are two obvious areas where memoizing can speed up invocations on distributed objects:

- *network objects*: each invocation on a network object requires opening and closing a network connection with a remote server. The dispatcher for network objects can memoize the state of the network connection.
- *oids:* each invocation on an oid requires a lookup of a network object from the locator. The dispatcher for oids can memoize the network object after the first lookup.

We can also modify the dispatcher for oids to optimize away network access for oids that point to objects within the local component. Efficiency was not a direct goal for our implementation; however, because we are able to create a clean design by employing flexible dispatching, applying the above optimizations is a straightforward task.

Thus, we have shown that by using flexible dispatching, we can allow for invocations on remote and foreign systems without changing DOM's basic model of computation. We presented two key elements of our distributed object implementation that utilize flexible dispatching, namely, network objects and oids.

3.4 Dispatching to objects in a database

Object-oriented databases preserve the structure of a program by retaining the identity and relationships of persistent objects. Different object-oriented databases employ different object models to accommodate their flavor of computation. With respect to dispatching, issues of integrating persistent objects are quite similar to those of distributed objects. To support integration of persistent objects, we must provide for:

- exporting references to persistent objects outside the database

- invoking operations on persistent objects from other systems. This invocation mechanism must support class-based and generic function models.

Hence, to provide access to objects in an object-oriented database, we can apply techniques similar to those described in Section 3.3. It is straightforward to represent the object-oriented database server itself as a network object. In this experiment, we elaborate on how flexible dispatching can be used to provide for the notions of identity and invocation in an object-oriented database. This generic invocation interface to an object-oriented database is effectively a *database object adapter* as described in [15].[7]

We can treat a database adapter as a network component, as long as it adheres to the required protocol. Recall that components must be able to export references and support invocations.

Exporting references. The database adapter interprets oids in the following way:

- the *component id* in the oid denotes a unique number referring to the database adapter's component id.
- the *object index* is a key into a mapping of indices to native database object references. When an invocation is forwarded to a database, we swizzle the oids in an invocation into native object references. We maintain the mapping of indices to native references within the database, so external references remain valid across sessions.[8]

Supporting invocations. To support invocations on objects within the database, we require a database adapter to support the invoke network message, just as we do for all network components. In our implementation, we used the dynamic invocation facilities provided by the particular object database that we were using. Upon the receipt of an network invocation, the database adapter:

- swizzles arguments of the invocation into native format. For basic data types, such as integers and strings, this amounts to a simple conversion. Oids are only valid if their component identifier matches that of the current adapter. Valid oids are swizzled into native
object references.

7. For this experiment, we used Ontos, a commercial object-oriented database management system which is built on top of the C++ object model. Ontos provides services typical of object-oriented databases, such as persistence and query capabilities, run-time typing, and dynamic invocation facilities.

8. This implementation of a mapping from oids to references is not very efficient. The identity mapping can be made more efficient by applying better search techniques (like hash-tables) or by having better knowledge of database's persistent identity criteria and providing a functional mapping instead. Since DOM does not make assumptions about the representation of object indices, we can change their representation easily.

- packages all the arguments into an argument list and calls the native dynamic invocation mechanism to process the argument list (causing an internal dispatch to take place within the database system)
- returns the result to the caller. If the result is not a basic data type, the adapter swizzles the resulting reference into an oid whose component id matches that of the database adapter.

In the absence of dynamic invocation facilities, handling the invoke messages could be accomplished by statically generating "wrappers" to take care of swizzling and unswizzling incoming invocations. (Such wrappers can be generated automatically.)

Thus we can transparently integrate objects belonging to object-oriented databases within the DOM environment. We have indirectly taken advantage of flexible dispatching in this experiment, since the bulk of our work here used the notion of an oid, and oids are implemented using DOM's dispatching mechanisms.

By creating an adapter that supports DOM's abstract notion of invocation, we have encapsulated the rest of the system from the details of the dispatching process that the object-oriented database uses.

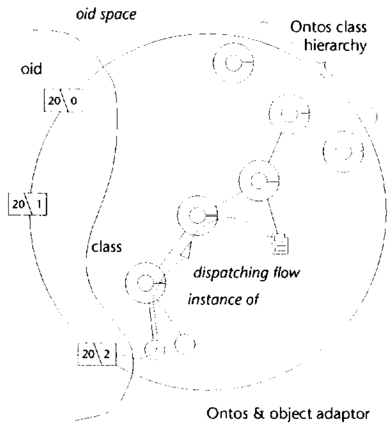

Figure 4. Dispatching to objects in an object database. Note the similarity of this interaction with the distributed object case.

3.5 Dispatching to rules in a rulebase

A traditional rule-based system uses a "closed" architecture, or at most includes an interface to a database system for retrieving data needed to process the rules. Recently, rules have started to play an important part in systems such as databases and programming languages, because rules tend to be a flexible method of specifying behavior. In this experiment, we explore how we can use DOM's flexible dispatching to allow for interoperability between the rule-based and object-oriented paradigms.[9]

The rule-based model. Traditional rule-based systems consist of two parts: rules and facts. The heart of a rule-based system is the rule engine, which takes the rules and facts and uses them to reach a particular goal or computation. For example, here is a fragment of a simple rule-based program that calculates weekly pay for employees based on their salary kind (annual vs. hourly):

```
;;; Depending on the kind of an employee asserted by this rule
;;; rulebase will fire either Pay-week-annual or Pay-week-hourly.
(defrule Employee-kind
    (employee ?enumber)
    (empl-kind ?enumber ?k)
=>  (assert (kind ?k)))

;; Pay-week-annual fires only for employees with annual salaries.
(defrule Pay-week-annual
    (kind "annual")
    (annual-salary ?s)
=>  (bind ?*weekly-pay* (/ ?s 52)))

;; Pay-week-hourly only fires for employees who are paid hourly.
(defrule Pay-week-hourly
      (kind "hourly")
      (worked-hours ?h)
      (hourly-rate ?r)
=>  (bind ?*weekly-pay* (* ?r ?h)))
...
```

The only way to provide data in a traditional rulebase is to declare facts. In practice, rulebases require access to large amounts of data, so many rule-based systems are integrated with database management systems. In such a system the user can write programs as rules in the rule language, and enter the data as tables within the database system.

Rules accessing distributed objects, rule system as a distributed object. To integrate rulebases in a distributed object system, we allow rule-based systems to access data that is outside the rule system. We provide a general mechanism for rules to make network invocations on distributed objects. In particular, rules can use the distributed invocation mechanism to access remote database objects. Finally, the rulebase system itself can be encapsulated as a distributed object.

Rules as methods of an object. To include rules as a part of the computation of a distributed object, we simply change the dispatcher for the object to dele-

9. For this experiment, we used C Language Interface Production System (CLIPS), a rule-based system available from NASA with support for rules and (foreign) functions, and Sybase, a commercial relational database system.

gate the computation to the rulebase upon certain invocations. For example, an employee object may use a rulebase to implement a computePay message. As discussed above, the rulebase can access a database in order to decide which rules to fire.

In this sense, the applicable rules are similar to methods in a traditional object-oriented language, like Smalltalk. At the same time, the data contained in the database can be thought of as the state of a Smalltalk object. Just as in the case for methods in Smalltalk, rules in our scenario access the data in the database *directly*.

Rules accessing data via methods of an object. We can improve the system design by restricting access to the state of an object, such as the employee object, to special accessor methods of the employee object. This arrangement gives the designer of the object control over how to manage the object's state [3, 11]. For example, the designer of the employee object may want to memoize the employee kind, rather than always looking it up in the database.

To implement this design, instead of passing the rule system a key to access data within the database, we modify the dispatcher of the object to:

- pass a reference to *itself* to the rulebase
- add accessor methods responsible for maintaining the data which the rulebase may access.

When the rulebase requires some state information, it can obtain it through an invocation on the distributed object. The interface to the rule system is simplified to include:

- an invoke operation similar to Obj.Invoke that allows the rulebase programmer to make invocations on DOM objects
- a self function that returns the identity of the managing object.

To use the new interface, we change the Employee-kind rule at the beginning of this section to:

```
(defrule Employee-kind
       (employee (id ?enumber))
 =>    (bind ?k (invoke (self) "kind"))
       (assert (kind ?k)))
```

For the self invocations to be implemented on top of a synchronous network invocation mechanism, we must allow for multiple threads to run through the distributed object. Otherwise the object cannot respond to an incoming accessor invocation while it is blocking for an outgoing network call to the rulebase.

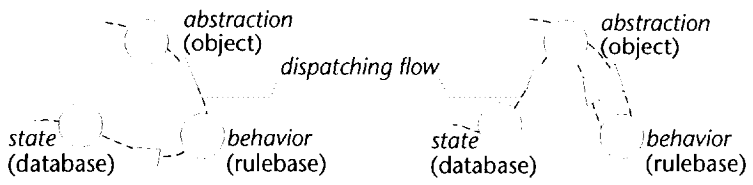

Figure 5. Two forms of interaction for accessing objects from a rulebase: Smalltalk-like interaction model (left) where state is accessed directly from behavior of an object and Self-like interaction (right) where state is accessed through callbacks to the object itself.

Hence, we have shown how we integrated rules as an effective method for implementing behavior in a distributed object system. Moreover, by relating our dispatching design to findings of recent object-oriented language designs such as Self [3], we were able to modify the dispatching process to provide more freedom to the object designer.

4 Related Work

We based our work on a broad collection of work done earlier in several areas. For a comprehensive survey of related developments, see [17].

The diversity of dispatching in object-oriented languages, especially Emerald [7], Self [3], Smalltalk [1], C++[2], and CLOS [4] prompted us to study variations of dispatching in different object systems. Each of these systems provides a different flavor of dispatching, yet all of them have a built-in method of dispatching.

CLOS Metaobject Protocol [10] and other work in meta-level architectures and computational reflection such as OpenC++ [18] provide "open implementations," allowing for applications to customize system services in a number of different areas, but they do not *directly* address the issues of interoperability across diverse object models, programming languages and paradigms. Techniques described in this paper can be applied readily in the context of a system with meta-level access to the dispatching process, such as [18].

Cecil [8, 23] provides a number of high-level structures on top of a classical static base, including ones related to dispatching. Cecil does not employ reflective techniques. Instead, its flexibility is designed by the language designer to cover all the conceivable high-level constructs, while allowing for static compilation as much as possible. We intend to explore how, by combining primitive notions in our object model, we can create higher-level constructs, like those of Cecil.

The OMG Common Object Request Broker Architecture (CORBA) [15] describes an architecture whose goals for interoperability are similar to those of our prototype. However, CORBA only supports a single object model, and requires other object models to be mapped to that one via an Interface Definition Language (OMG IDL). CORBA does not address implementation issues directly. Instead, implementation issues such as dispatching are handled by "object adapters" for different systems. In this sense, our flexible dispatching design can be used as a basis for building object adapters in CORBA.

Perhaps the closest match for DOM in CORBA systems is the IBM Distributed System Object Model (DSOM) [19]. DSOM is a distributed extension of the System Object Model (SOM) [20, 21] which allows for sharing libraries across language boundaries by employing an explicit meta-architecture. As both systems need to address dispatching issues in different models, our work is similar to DSOM in the context of dispatching. We address coarse integration of diverse object models, while DSOM focuses on a tight integration of more popular classical object models.

The design of the DOM scripting language was inspired by Scheme [6]. Obliq [22] and DEC SRC's Modula-3 network objects implementations [13] have some goals in common with DOM, for example in building a lightweight interpreted language for network objects. However, our approaches are complementary: Obliq and SRC network objects emphasize efficiency, while we aim at achieving flexibility.

5 Lessons Learned and Open Issues

Some of the lessons learned and interesting issues which we encountered in our experimentation effort were:

Modelling invocations explicitly. Our initial design split the notion of an invocation into two parts, a *receiver* and an argument list to be sent to the receiver. Often it is convenient to treat the whole invocation, including the receiver, as an explicit entity.

Call-space problems. There are a few places where our abstract notion of invocation breaks down:

- A message sent to a proxy can be construed in two ways: either a message sent to the proxy object itself, or a message sent that needs to be forwarded to the object which the proxy represents.
- Good-citizen messages are messages that all objects in the system support. For example, objects respond to a printString message by returning a printed representation of themselves. What happens if we send a printString message to the closure that results from evaluating (LAMBDA (x) x)? We must decide either to follow the printString convention, and return something like "#[function]" or apply the closure and return the result.

A number of generalizations can be made in order to accommodate the call-space problems. For example, we can make the different call-spaces explicit by allowing for a notion of context (system vs. user) in an invocation. Or we can allow for the same object to have multiple dispatchers. We need to examine the implication of such changes in the design.

Concurrency control. The experiments described here do not address concurrency control problems that take place in large distributed systems. We are currently investigating how to modify the dispatching process to accommodate different concurrency control mechanisms like those described in [24]. Implementing pessimistic control using locks similar to [18] is straightforward within our dispatching framework. Optimistic concurrency control is more challenging.

Metaobject Protocols. We intend to organize the meta-level access to the system that the dispatchers provide by using MOP techniques similar to ones described in [10,18]. As we noted earlier, programming with explicit dispatching is similar to programming with explicit "goto" statements: its flexibility is both useful and dangerous. After understanding the useful and common ways of dispatching, we can use MOP techniques to construct "higher-level" abstractions.

Separation of meta- and base-levels. Dealing with a mix of meta- and base-level calls to the same object – for example, meta-level calls to add methods to the generic function (draw 'addMethod aMethod) vs. base-level client invocations on the generic function (draw aRectangle aDevice) – tends to confuse the programmer. Part of the confusion is due to the fact that we use the same mechanisms (dispatchers) to allow access to both meta-level notions and base-level notions.

Optimization. We intend to examine the performance implications of our design. One advantage of our design is that its performance model is localized: unlike systems with one fixed way of dispatching, we can allow for one form of dispatching to be optimized without affecting the rest of the system.

6 Summary and Conclusion

In this paper, we reported on a series of experiments in a distributed object system that show how a flexible notion of dispatching can be used to integrate objects belonging to different models, systems, and paradigms. Experiments reported in this paper covered a wide range of concepts including:

- composition facilities such as classes with inheritance, and generic functions
- distributed systems concepts such as client-server computing and distributed objects
- object-oriented databases and rule-based systems.

In all cases, flexible dispatching provided us with a powerful framework for integrating diverse systems. Using this framework, we have integrated code written in a wide variety of systems and models successfully, including: Modula-3, C, C++, Macintosh Common Lisp, CLIPS rule-based system, Sybase relational database, and Ontos object-oriented database system. Since DOM enforces few restrictions on object implementations, we can easily extend a programming language or system to support DOM.

Dispatching is only one of the ways which object systems differ. We intend to experiment with other differences – for example, the treatment of object identity, state, and life-cycle – under the charter described in [25].

Acknowledgments. Farshad Nayeri and Joe Morrison implemented the core of DOM-3 based on discussions with other members of Distributed Object Computing group at GTE Labs, especially, Frank Manola, Sandy Heiler, and Mark Hornick. Ben Hurwitz joined the project later, and implemented a number of the experiments described in this paper. Emon Mortazavi artistically created some of the illustrations.

Geoff Wyant, Gail Mitchell, Sandy Heiler, Joe Morrison, Cristóbal Pedregal Martin, Luca Cardelli, Dimitrios Georgakopoulos, Mark Hornick, Michael Brodie, and Lauren Schmitt reviewed earlier versions of this paper and provided us with many insightful comments. We hereby thank all who helped in making this idea become reality.

References

1. A. Goldberg and D. Robson, *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley, 1983.
2. B. Stroustrup, *The C++ Programming Language*, Addison-Wesley, 1986.
3. D. Ungar and R. B. Smith, "Self: The Power of Simplicity", in [26].
4. G. Steele, Jr., *Common Lisp: The Language, Second Edition*, Digital Press, Bedford, 1990.
5. Greg Nelson, ed., *Systems Programming with Modula-3*. Prentice Hall, Englewood Cliffs, NJ, 1991.
6. W. Clinger and J. Rees, ed., "Revised^4 Report on the Algorithmic Language Scheme", ACM Lisp Pointers IV, July-Sept. 1991.
7. A. Black, N. Hutchinson, E. Jul, and H. Levy, "Object Structure in the Emerald System", in [27].
8. C. Chambers, "Object-Oriented Multi-Methods in Cecil", *Proc. of the 6th European Conference on Object-Oriented Programming*, Springer-Verlag, 1992.
9. D. H. H. Ingalls, "A Simple Technique for Handling Multiple Polymorphism", in [27].
10. G. Kiczales, J. des Rivieres, and D. G. Bobrow, *The Art of the Metaobject Protocol*, MIT Press, Cambridge, MA, 1991.
11. *Dylan, an Object-Oriented Dynamic Language*. Apple Computer, April, 1992.
12. A. Tanenbaum, "Distributed operating systems anno 1992. What have we learned so far?", in *Distributed Systems Engineering*, Vol. 1, 1993.
13. A. Birrell, G. Nelson, S. Owicki, E. Wobber "Network Objects", *Proc. of Symposium on Operating Systems Principles, 1993.*
14. D. Maier, J. Stein, A Otis, and A. Purdy, "Development of an Object-Oriented DBMS." in [27].
15. Object Management Group, *The Common Object Request Broker: Architecture and Specification*, OMG Document Number 91.12.1, Rev. 1.1, 1991.
16. M. L. Scott, "Messages vs. Remote Procedures is a False Dichotomy", ACM SIGPLAN Notices 18(5), 57-62, May 1983.
17. F. Manola and S. Heiler, "A 'RISC' Object Model for Object System Interoperation: Concepts and Applications", Technical Report 0231-08-93-165, GTE Laboratories Incorporated, September 1993.

18. S. Chiba and T. Masuda (1993) "Designing an Extensible Distributed Language with a Meta-Level Architecture", *Proc. of the 7th European Conference on Object-Oriented Programming*, Springer-Verlag, 1993.
19. F. Campagnoni, "The Distributed System Object Model: IBM's Object Request Broker Implementation", position paper to *CORBA Implementors Workshop*, June 1993.
20. N. Coskun and R. Sessions, "Class Objects in SOM", *IBM Personal Systems Developer* (Summer 1992): 67-77.
21. R. Sessions and N. Coskun, "Object-Oriented Programming in OS/2 2.0", IBM Personal Systems Developer (Winter 1992): 107-120.
22. L. Cardelli "Obliq - a Lightweight Language for Network Objects", unpublished DEC SRC Technical Report, November 1993.
23. C. Chambers, "The Cecil Language: Specification and Rationale", Technical Report 93-03-05, University of Washington, March 1993.
24. D. Georgakopoulos, M. Hornick, P. Krychniak, and F. Manola, "Specification and Management of Extended Transactions", *Proc. of 10th International Conference on Data Engineering, 1994.*
25. F. Manola and S. Heiler, "An Approach to Interoperable Object Models", in M. T. Özsu, U. Dayal, and P. Valduriez (eds.), *Distributed Object Management*, Morgan Kaufmann, 1994.
26. N. Meyrowitz, ed., *OOPSLA '87 Conference Proceedings*, ACM, Oct., 1987, published as *SIGPLAN Notices*, 22(12), Dec., 1987.
27. N. Meyrowitz, ed., *OOPSLA '86 Conference Proceedings*, ACM, Sept., 1986, published as *SIGPLAN Notices*, 21(11), Nov., 1986.