Distributable Functional Objects

John Corwin Ryan Lee Binh D Vo Tom Woodfin
{jcorwin, ryanlee, bdv, twoodfin}@mit.edu

December 6, 2001

Abstract

This paper describes a distributed object system that abstracts method calls on an object interface from both the object's physical location and the object's distribution strategy. The system will greatly simplify the creation of a multitude of different systems. We include a few examples of common systems that are both simpler and more reliable because of the use of distributed functional objects, compared with traditional network object systems.

1 Introduction

Traditional distributed object schemes have been extremely successful at abstracting the difference between local and remote method calls for the programmer. However, these schemes often restrict the flexibility of the system designer by implementing a single mechanism for the distribution of objects. The limitations of this one-size-fits-all approach cry out for a separation of mechanism and policy. This is what we have attempted with Distributable Functional Objects (DFO). The DFO system allows the independent development and subsequent combination of implementation objects and server and client distribution policies. The result is a flexible system that allows distributed systems designers to design and select distribution policies that fit the needs of their system.

Given the time alotted for this project, our system could not afford substantial complexity in either design or implementation. On the design side, the work done in Network Objects [1] was extremely helpful, and while our design deviates from Network Objects in several respects, its emphasis on simplicity made reproducing much of its functionality easy. The implementation was similarly simplified by our use of the Objective Caml (OCaml) language [3], a member of the ML family with object-oriented features and an optimizing compiler.

Our two most significant extensions to the Network Objects model revolve around the client and server policies and the addition of policy information to the object protocol. The goal in both cases is added flexibility. Client and server policies manage the distribution of distributable objects from the client and server ends, respectively, while the policy information attached to the object protocol allows the policies on both ends to communicate and negotiate effectively in a manner independent of the type of object they are managing.

These extensions permit a multitude of design options. Client policies can maintain multiple references to servers providing the desired object. Server policies can redirect service requests to another server. Client and server policies can communicate to coordinate activities such as object replication and mobility.

Many of these design options are simplest when using purely functional objects, the "F" in DFO. Functional objects can be replicated as needed without concern for consistency, and can be recreated with only their intialization parameters. OCaml provides full support for functional objects, and we used them extensively during the development of the system and our sample applications.

2 Related Work

Network Objects

As discussed in Network Objects [1], the Network Objects system allows a program on a local system to make a method call on an object in a remote system just as if the object were locally present. Through the use of a stub generator, which produces the necessary client and server interfaces, as well as an abstract notion of a transport, the implementation of the object exists on the server system, while a client stub object provides the means for a client program to call methods on the server object. A global object reference count allows for garbage collecting most inaccessible network objects.

Although Network Objects does abstract the *physical* means by which objects are distributed through its `Transport.T` type, the system itself is heavily weighted toward a single, immobile instance of a server object and a standard means of locating and calling methods on that object via the "surrogate." Although the authors mention the possibility of performing "tricks" such as custom transports, for implementing such features as replication, particular characteristics of their implementation, such as the basic wire representation of an object including its "owner space" would make these tricks difficult, and certainly not straightforward.

Java RMI

As its name suggests, Java Remote Method Invocation [4] is the standard means by which Java objects can expose their methods to external virtual machines, possibly over the network. Semantically, Java RMI is quite similar to Network Objects, in that these remote calls are indistinguishable from local calls, with the exception that error conditions due to network problems can arise on remote calls that would not arise for local calls. Most relevant to this discussion is Java RMI's bootstrapping mechanism, which takes advantage of a naming service to retrieve an initial remote object. Our system takes a similar approach.

Even moreso than Network Objects, Java RMI is inflexible in its approach to object distribution. An object exists in the memory space of a virtual machine, which also exposes an RMI service for mediating remote method calls. There is no facility for replication or object migration.

3 Design

Ultimately, the design of a distributed object facility in the style of Network Objects or Java RMI is all about what goes on under the well-understood abstraction of the method call. Java RMI and Network Objects offer a mechanism to package up the call and its arguments, then transmit the package to a server, which unpackages the call and applies it to an implementation object. The result returned by the actual method is similarly packaged and transmitted, then unpackaged at the client end and returned to the user. However nothing in the semantics of a method call necessitates this approach.

In fact, in many situations it might be sub-optimal. Suppose we are designing a distributed primality tester. The interface we will present to clients is a simple method call which takes an integer as a parameter and will return a boolean value signifying whether the integer is prime. Rather than a single implementation object existing on a single server, we could achieve much faster results with one or both of multiple implementation types and multiple servers. One implementation type might perform a fast algorithm that does not always succeed in determining a result, while another might perform a slower but result-guaranteed algorithm. Some implementations might cache results or precompute the results of expected requests. A client implementation could send a request to multiple servers, returning the first result received. It might keep track of those servers that are returning results quickest, perhaps due to low load, and favor sending requests to these servers in the future.

The key requirement is flexibility. We cannot imagine all the requirements of future system designers, and even if we could, incorporating mechanisms to support all possible designs would be a Herculean task. Rather, we choose to provide a flexible infrastructure allowing the system designer freedom to choose a distribution strategy that matches her application.

On the other hand, we do not want this flexibility to come at the cost of unnecessary complexity for either the system designer or the application programmer. Simple designs should be simple to implement. On the application side, we essentially retain the model of Network Objects and Java RMI. Applications call remote objects using normal method call semantics. For the system designer, it is trivial to implement the standard "single-server" approach to remote method invocation (we have done so ourselves for several of our examples).

More complex designs depend upon the abstraction provided by client and server policies. These policies ultimately determine how a method call and its response are handled. Because client policies exist (naturally enough) on the client, while server policies exist on the server, the system provides a standard way for the two to communicate through the exchange of policy information. Policy information is represented as a mapping between strings, allowing arbitrarily complex exchanges of both strings and properly marshalled objects.

Distributable objects

At its core, a distributable object is merely an identifier. For the identifier to have any meaning, at least one DFO server must associate it with a server policy, and at least one client application must have a reference to it encapsulated in a client stub. When a DFO server is presented with an identifier it recognizes by a client, it maps the identifier to an object that implements a server policy. The server policy is then responsible for dispatching any method calls made on the distributed object and re-

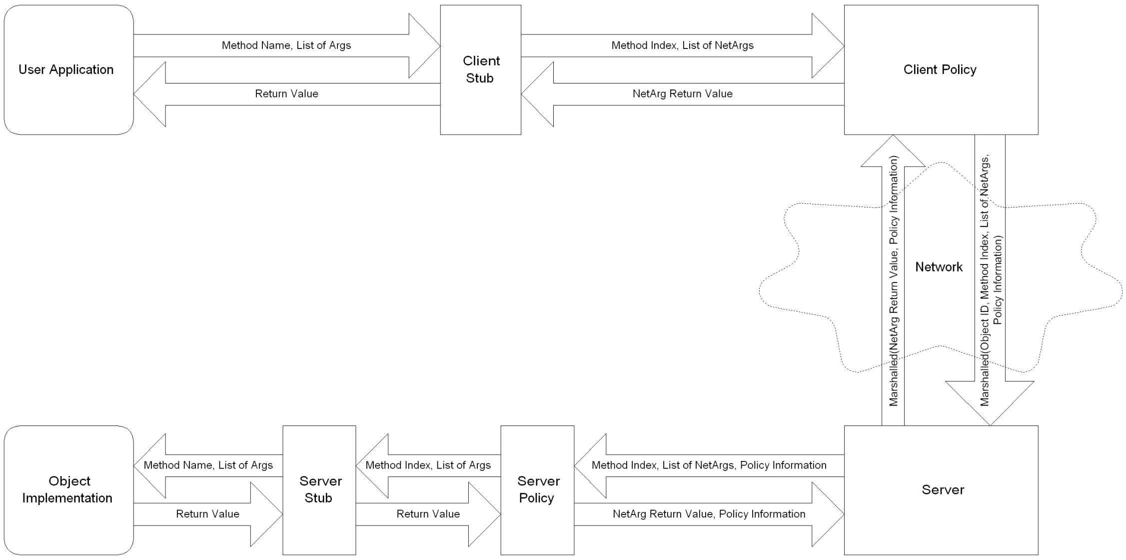

Figure 1: The DFO Architecture

ceived by this DFO server, and for sending the result of the method call back to the client machine. Figure 1 shows an overview of the system's architecture.

Client stub

The client stub implements the interface of an associated distributable object class. Rather than containing an actual implementation of the functionality of the object, or directly marshalling and transmitting the arguments of the call to a server, the client stub merely packages the arguments to the method call as well as an index specifying which method was called and the identifier of the distributed object into a tuple, and passes the tuple on to the client policy associated with the distributed object. When the client policy returns a distributable object, the client stub is responsible for instantiating a new client stub to be used to call methods on that object.

Client policy

The client policy presents three methods via its interface. Two are associated with garbage collection and will be discussed along with the implementation. The third is the `process` method. This method is free to do anything it likes to process the method call it receives from the client stub, however it must return an object type expected as the return value of the method call. Different approaches to client policy will be discussed in the Examples section (5).

DFO server

The DFO server listens on the network for two kinds of requests: method requests and service requests. Service requests are used to initially bootstrap the system. A globally unique identifier is sent by a client to the server and if the server recognizes the identifier as being associated with a particular distributed object type, it returns a corresponding initial distributed object identifier along with some policy information. Method requests can be called on this or any other distributed object. Method requests include an identifier indicating the distributed object the call was made on, as well as a collection of arguments, some with their own policy information collections and an global collection of policy information. A server policy is located by looking up the identifier in the DFO server's table, and all the information associated with the method call is passed along to the server policy for processing. The result of this processing (discussed below) is then marshalled back over the network.

Server policy

The server policy is responsible for processing method calls. As with the client policy, its most important method is `process`. In the trivial case, `process` simply passes the index of the method as well as the arguments through to the server stub, which in turn calls the appropriate method on an implementation object. However, to facilitate more complex processing, each object argument, as well as the entire call, is associated with a collection of policy information. This information can be used to locate an object not currently on the server, to indicate whether a response can or should be cached, to migrate the object, or for any other purpose. It is entirely up to the policy how to interpret the policy information.

The result of the `process` method is a union representing marshalled return value of the method call (including a policy information collection if it is a distributed object), as well as a policy information collection associated with the entire result. This might be no information at all if the call is unexceptional. To follow up on our previous example, it might indicate to the client that the response was indeed cached. The server policy might want to inform the client policy that the distributed object has been moved to another server, and rather than use the server's built-in redirection, it can use the (included) address of the object's new home.

Server stub

The server stub has a single method called `dispatch` that takes as parameters a method index and a list of marshalled arguments. `dispatch` simply unmarshalls the arguments and passes them to the appropriate method of the object implementation the stub is associated with. It then marshals and returns the result of this method call.

Object implementation

The object implementation need only be an implementation of the same interface as the client stub, and can be written without any special knowledge of the DFO system – it does not have to support any additional interfaces or provide any extra functionality to work as a distributed object.

Policy information

Policy information is represented by a string to string mapping. The use of a general mapping has the advantage that extensions to policy protocols can be made easily without breaking older implementations. For example, if a server policy adds caching support, it can add new mappings to the policy information returned such as "isCached"-> "true". Old clients that are unaware of caching can proceed as before, extracting values from the policy information they expect, while new clients can access the additional information.

Also, the use of strings along with the support of a good marshalling library (such as that provided by OCaml), allows (nearly) arbitrary types to be included in the policy information collection.

4 Implementation

All code for DFO was written in OCaml, aside from a small interface to the C library we used (`libuuid`) to generate object identifiers. After some initial design work, it took our team of four programmers about a week to build DFO and to make example applications based upon it.

About OCaml

OCaml is self-billed as "the programming tool of choice for discriminating hackers." It combines functional, imperative, and object-oriented programming and employs a strong, static type system that can detect many common programming errors at compile time. OCaml also supports type inference, polymorphic types, higher order functions, and closures, all of which contribute to concise code and a good deal of flexibility in choosing abstractions.

As an illustration OCaml's power, we were able to implement a majority of the functionality of `libasync`, an asynchronous socket library written in C++, in less than 80 lines of code. Furthermore, the language guarantees that this code will never go wrong as a result of a NULL-pointer or a bad object cast.

DFO was tested on x86 Linux machines, as OCaml is developed on and readily available for that platform. OCaml is also available for Windows, Macintosh, and on most Unix systems, meaning DFO should be portable between several different operating system and processor environments.

Client and server stubs

The client and server stubs are straightforward, repetitive modules that handle the marshalling and unmarshalling of method arguments and return values. Each object interface exposed through our system requires a client stub and a server stub. Due to time constraints, we were unable to write a full stub generator; all our stubs are written by hand. This was something of a tedious task, and given more time, we would like to have implemented an automated stub generator instead of introducing more possibilities for human error.

Distributable objects

We dealt with the problem of uniquely identifying objects throughout DFO by using the `libuuid` library to generate object identifiers. The identifier given to an object at its instantiation is reasonably assured to be unique both within a system and across all systems. The library is failry standard, and linking C code with OCaml is simple enough, so we rely on the `libuuid` generator to assign objects identifiers.

To make passing arguments between distributable objects feasible, we defined a `net_arg` type for use inside stubs. The `net_arg` type is a union type representing a marshalled value or a remote object identifier. The stubs then rely on marshalling and unmarshalling a `net_arg` to determine the actual argument type.

DFO server

The DFO server receives calls from the client-side and passes them to the appropriate server policies. To do this, it keeps two hash tables: one for matching service IDs to server policy objects, and one for matching object IDs to server policy objects. The first table is necessary for keeping track of available services and instantiating new objects, and the second for matching IDs for method calls. This layer is very simple, it needs only lookup up server policies to pass calls to. Most of the important decisions are made by the policies.

Client and server policies

The client policy is responsible for handling calls received from the client stub. It must contain three methods: `process`, `send_release`, and `send_addref`. `process` is called by the client stub, and returns an appropriate data type for the specified call. Although usually it should use the marshalling code to relay the call to the server-side, it is free to handle it in any way that the policy writer wishes to. `send_release` and `send_addref` are used to support network garbage collection and are called when an object becomes locally unreachable and when an object is passed as an argument to a method call, respectively.

The server policy receives calls from the client policy and handles them. This may involve simply passing them along to the server stub which calls corresponding methods in the object implementation, computing simple results on its own, caching previous results, forwarding to other servers, or whatever else is deemed appropriate by the policy. It also will receive `addref` and `release` calls from the client policy, which it will handle as appropriate.

We implemented reference-counting network garbage collection because of its simplicity, and, given full-featured garbage collection for local objects, should work well for most systems. Reference counting in this case simply keeps track of the number of client machines that are using an object. This gives the sever policy information on whether to inform the DFO server that this object is no longer needed, and can thus be locally garbage collected.

We provide a set of standard server policies, which may be inherited from to create any number of systems. The server policy may inherit from a refcounted policy or a nonrefcounted policy, a policy that takes a single host or a list of host servers, or can have its own specially designed policies. The included set of policies are sufficient for simple systems, such as standard single-host forwarding policies, but are by no means a limit on what can be done.

Bootstrapping

On the server side, a server is created and services instantiated by functions in the `NetObjServer` package. The `create_server` function creates a server on a given host and port. The `register_service` function maps a unique identifier to a server policy.

A client bootstraps by connecting to the server and transmitting a `ServiceRequest` containing the service identifier. The server will respond with a `ServiceResponse` containing a `net_arg` and a collection of policy information corresponding to the object managed by the server policy. Services are intended to be "static" objects containing sufficient methods to retrieve through standard `MethodRequests` whatever the client might require. The `file_system` object in our network file system (see Examples in section 5) is such an object, and is provided as a service. Clients retrieve the file system through a `ServiceRequest` but subsequently browse, retrieve and manipulate the files through `MethodRequests`.

Error handling

An important benefit of a client policy mediating method calls is the potential for intelligent error handling. Unlike Java RMI [4], a network failure resulting in the inability to contact a server does not automatically result in an error condition. A sufficiently intelligent client policy could (for example) contain references to multiple servers, and refer the method call to a second if the first fails.

Our system provides a single exception type, `Net_obj_error`. `Net_obj_errors` are thrown by client policies when either network failures occur which fall outside the ability of the policy to handle, or if the policy information returned by the server is incomplete or incomprehensible to the client policy.

5 Examples

To demonstrate DFO's power, we implemented four systems which each use different client and server policies. NetFS implements a simple file transfer utility, Crack performs a distributed key space search, Distributed Stock Ticker shares stock prices across multiple servers, and NetDFS represents our entry for the class of distributed, read-only file sharing programs.

5.1 NetFS

NetFS is a simple demonstration of DFO's functionality. It provides a simple FTP-like client interface to a file server. As a first-pass application, its server and client policies were kept as basic as possible.

NetFS Operation

Being a basic implementation, the NetFS server runs out of whichever directory it is invoked in. NetFS clients can then connect to the server to perform simple the operations of `ls`, `cat`, `get`, and `put`, equivalent to the FTP syntax for listing a directory, reading a file, and transferring files from and to the server.

NetFS Policies

To illustrate simple policies, NetFS makes use of the most basic server policy. That is, the process method passes the arguments to the server stub, which then invokes the proper object method.

The basic client policy is to simply take the requested file operation and pass it on to whichever file server is connected to.

5.2 Crack

Crack is a distributed key search, making use of brute force technique to discover the key used in encrypting known cleartext into known ciphertext. The encryption method being attacked is the Data Encryption Standard (DES) [5], an encryption algorithm which has previously been defeated through distributed efforts [6].

Background of DES

DES was formerly a U.S. standard for secure encryption, though repeated demonstrations of DES' increasing weakness to brute force attack motivated the government to adopt standards with larger keyspaces [7].

The algorithm uses 56-bit keys, in effect, and an intentially complicated and confusing permutation algorithm to encrypt information in 64-bit blocks. While derivative mathematical attacks exist, they are not much more effective than brute force attacks.

DES was chosen for two reasons. Since DES has been under scrutiny for more than 20 years, ample documentation exists, making its encryption and decryption implementations for this project fairly straightforward. Also, attacking an algorithm known to be defeatable by brute force methods allows for some comparison, given that sufficient resources exist to test Crack, as well as a guarantee that the problem is indeed solvable.

Crack is not intimately tied to attacking DES, and one future improvement may be to simplify changing which algorithm's keyspace Crack is brute forcing.

Crack Operation

Crack is currently tied to a key search without attempts at actual decryption. The correct clear text is required to compare each key's decrypted ciphertext against. Crack was implemented this way because defining a reasonable metric of whether or not a certain key produces interesting decryption results is a problem unto itself. Adding such a metric would, however, make Crack far more interesting. As it is, Crack is still useful for discovering keys which may have been reused.

Crack is composed of client and server software. A user runs the Crack server as a background process and invokes the client to find a key.

Crack Policies

The Crack client and server policies maintain a list of possible servers which a client can contact. The Crack client policy then forwards the initial request to all available servers. Starting with the smallest key, it then loops through the entire keyspace, keeping known servers busy. This policy is consistent with Crack's compute-intenstive nature.

Crack Performance

In terms of key cracking, Crack could probably use some performance tweaks to increase its keyrate. At its current rate of operation, without considering other factors, 100,000 Crack servers would take 90 years to go through a quarter of the available keyspace, something the latest distributed effort did in 22 hours.

5.3 Distributed Stock Ticker

Distributed Stock Ticker is a system for relaying stock prices to multiple target servers. The server side is responsible for receiving messages from a client, forwarding them to all servers, and ensuring that they are displayed on all servers in the same order. In this version, servers register subscriptions to stocks they are interested in displaying as well as information about which stock quotes they will forward to other servers on startup.

Previous Work

The previous version of Distributed Stock Ticker was coded in C++ using an asynchronous library and a limited RPC system. The new version is much simpler because of the interfaces supplied by DFO and because of our use of OCaml. DFO provided abstractions to make the details of communication transparent to the rest of the application.

DFO Stock Ticker Details

Since the servers are now represented by objects, the details of communication are made transparent to the client-side. The client application does not need to consider timeouts for the messages, or know the address of servers that are interested in their stock updates. All messages are submitted as simple method calls to the remote server object.

Because of the provided marshalling system, neither the client nor the server has to include a specially designed formatting for the messages. Different requests can be handled by different methods, and multiple components of the requests can be sent as different arguments.

And, most importantly, the addition of a flexible set of client and server policies made it easy to tailor the network objects communication systems to our needs. Whereas an existing network objects system would require us to adhere to a pre-defined policy, with DFO, we were able to create our own policy to optimize network traffic and handle subscriptions.

5.4 Distributed NetFS

To demonstrate how policies can modify the behavior of a system, we implemented NetDFS, a distributed, read-only filesystem. To create a distributed file server, we only had to modify the NetFS policies for file and filesystem objects and make some small changes to the server code. The result is a system that would be useful for applications like accessing a collection of mirrored distribution servers.

NetFS Policies

Both the server and client policies were extended to support keeping track of a list of servers providing access to a file system. We updated the server code to support reading a list of servers that form the distributed file system and modified the file system object's server policy to send a list of available servers to connecting clients. Then, we just had to change the file system's client policy to select a random server from its server list when requesting new file objects, and to be able to switch to a different server if it encounters an error while performing an operation on a file.

NetDFS Operation

NetDFS operates just like NetFS, except that requests can go to any file server in the system and writing to a file is not permitted. Creating new policies for the file system was relatively simple, and adding other extensions such as prioritizing the servers or allowing for severs to dynamically enter and leave the system could also be done fairly easily.

6 Future Work

For a rapid prototype written in a week, DFO and its example applications work remarkably well. However, such a short development schedule imposes some constraints on the completeness of the system.

We currently make use of OCaml's marshalling module to send objects across the network. However, the marshalling module is not type-safe, and the sender and receiver must agree by contract on the types of arguments and responses. To ensure type-safe network transactions, it may be better to improve or replace the marshalling module with one that is type-safe.

As mentioned above, we would have liked to implement an automated stub generator.

To further ease development using DFO, we would like to create a standard library of policies, such that creating a policy becomes a simple task of matching components of existing policies to the task at hand, extending them if necessary through inheritance schemes.

7 Conclusions

The DFO system is easy to use, and greatly simplifies the creation of distributed object-oriented systems. It abstracts from the object interface the details of object locations, garbage collection, and distribution strategies. It also provides components that facilitate the creation of new distribution strategies, as well as the use of existing simple strategies.

As was shown, DFO made the implementation of some existing non-trivial systems easy. Many example systems were created in a matter of days, and have remained stable with a minimum of revision.

References

[1] Andrew Birrell, Greg Nelson, Susan Owicki, and Edward Wobber. *Network Objects*. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 217-230, December 1993.

[2] Guy Cousineau and Michel Mauny. *The Functional Approach to Programming*. Cambridge University Press, Cambridge, 1998, ISBN 0-521-57681-4

[3] The OCaml Language.
<http://www.ocaml.org>

[4] Java Remote Method Invocation.
<http://java.sun.com/products/jdk/rmi>

[5] J. Orlin Grabbe. *The DES Algorithm Illustrated.*
<http://www.aci.net/kalliste/des.htm>

[6] Matt Curtin, and Justin Dolske. *A Brute Force Search of DES Keyspace.*
<http://www.interhack.net/pubs/des-key-crack/>

[7] National Institute of Standards and Technology. Advanced Encryption Standard.
<http://csrc.nist.gov/encryption/aes>
December 4, 2001.