Stanford Digital Library Interoperability Protocol

Stanford Digital Library Group
{hassan,paepcke}@cs.stanford.edu
http://www-diglib.stanford.edu

1 Introduction

This document describes and specifies Stanford's Digital Library Interoperability Protocol (DLIOP). This initial version of the protocol specifies interactions that allow clients to request and asychronously receive information from other objects that in turn may access information services on the Internet. All components involved in the protocol are objects distributed arbitrarily across machines, possibly of varying computer architectures, and potentially written in different programming languages.

An example of a simple arrangement is shown in Figure 1.

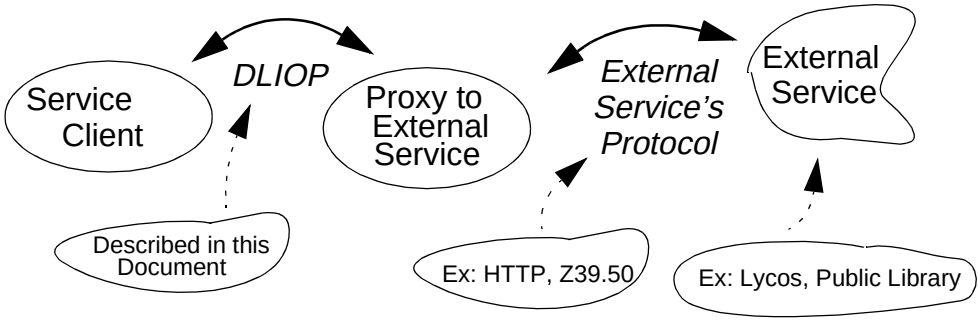


Figure 1: Focus of this Document (Ovals are objects)

The main goal of a DLIOP interaction between a client and an external service proxy is to submit a query to the proxy, have the proxy interact with the external service to retrieve information, to have that information transferred to the service client, and to have the results presented there as objects.

What is special about the DLIOP is that it makes use of CORBA's distributed object technology and can therefore more easily serve as a testbed for protocol features other protocols cannot address as easily.

For example, the protocol combines advantages of stateless protocols

such as http (e.g. optimization for short-lived resource usage) with the advantages of session-based protocols such as Z39.50 (e.g. interaction efficiencies for multiple, related interactions)[ansi95].

The distributed object capabilities afford use of the DLIOP as a testbed for experimenting with dynamic resource reallocation to achieve load balancing. The protocol allows documents and computation to be moved among machines while interactions between clients and services are in progress.

The protocol also enables experimentation with different caching strategies in contexts where clients sometimes request follow-up information quickly, sometimes after several days have elapsed.

A more descriptive motivation and explanation of rationale is provided in [paep96a]. In the following section we present an overview of the protocol. In section Section 3 we go through each method and explain the parameters. The appendix contains the COBRA Interface Definition Language (IDL) specifications of the protocol.

2 Overview

The protocol specifies methods invoked among two basic kinds of components: Client objects, and collection objects. When we speak of ‘objects’, we always use the term in the sense of object-oriented programming. Therefore, by client objects we mean objects that implement some client application attempting to access information. Furthermore, the objects that are players in the interop protocol are distributed CORBA objects. This means that even though they invoke methods on each other, they may reside on different machines and may be implemented with different programming languages.

CORBA interfaces do not include object instance variables. Instead, a separate ‘Property Service’ is defined for attaching properties to objects, for accessing them, deleting, etc. When we speak of object properties, we mean the CORBA Property Service facilities.

Collection objects conceptually hold result objects which contain the infor-

mation clients are looking for. These could, for instance, be objects containing the text of documents. Collections have a simple interface, which includes methods such as **`GetTotalItems()`** which returns the number of items in the collection, or **`AddItems()`** which puts new items into the collection if possible. In Figure 1, the proxy representing the external service is implemented as a special kind of collection object, a contrainable collection which is a collection that may be queried.

Constrainable collections are an important subclass of collections. They may be asked to produce a result collection which contains a subset of their contents. We use these constrainable collections as proxies for information providers out on the net. We can do this by allowing collections that do not really contain any objects, but that 'pretend' they do. Then, when such a collection receives a request to constrain itself, it issues a query to the outside information provider, retrieving information from it. This information, which is usually not in the form of objects, may then be materialized into objects and held in the collection for delivery to the client.

2.1 Basic Search Interaction

Figure 2 shows an example of a basic search interaction. The client wishes

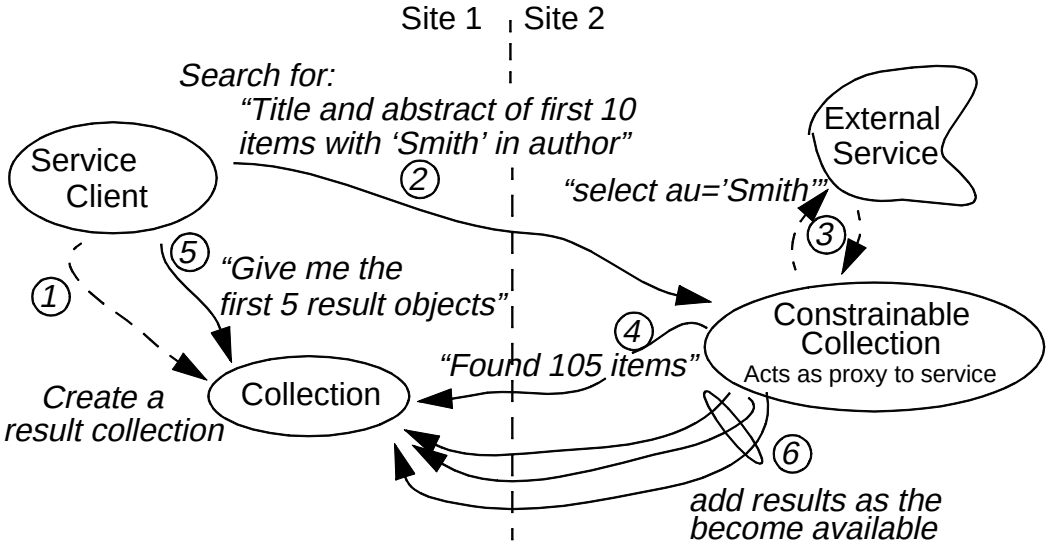


Figure 2: A Basic Search Interaction

to search over some external service for which a constrainable collection

object acts as a proxy. Note that while the steps in the figure are numbered for reference, most of them occur asynchronously. Arrows in Figure 2 represent method calls.

As a first step, the client creates a local collection object which will hold the results. Then it issues a query to the constrainable collection which acts as the external service's proxy. The query contains information on which objects are to qualify in the query (authors must contain the word 'Smith'). It also contains instructions on how many results should be returned as soon as they become available (10), and which parts of those results should be included (only abstract and title of each result). In addition, the query request includes a pointer to the client's local result collection. The proxy collection will use this pointer to deliver results.

Once the query has been delivered, the client is free to perform other work while the query is processed: the query call to the proxy collection is asynchronous. Alternatively, the client may immediately ask its local result collection for the total number of results, and/or for some or all of the result objects. These calls could block until the required information is actually available.

Meanwhile, the proxy collection delivers the query to the external service by whatever means are appropriate: http, Z39.50, a telnet connection, its local file system. As soon as the proxy collection knows how many hits to expect, it notifies the client's result collection. In (possibly) multiple calls to the client's collection object, the proxy collection subsequently delivers results. Each call (step 6) to the client's collection delivers some number of title/abstract values as lists, each list being the desired (title/abstract) excerpt from one result. The client collection creates a local object for each result, filling its title and abstract properties with the corresponding values[1]. It is up to the proxy collection to decide whether to wait for all 10 results to arrive from the external service before delivering them to the client's result

1. Note that DLIOP could have had the proxy collection materialize the raw information into objects. Pointers to these objects would then be passed to the client. This would have had two disadvantages: the proxy collection would have had to maintain these objects indefinitely, and every object property access by the client would have involved a remote method call.

collection, or whether to collect a few, and to deliver them as early as possible.

2.2 Getting More Result Objects of the Same Query

At some point, the client will have received and examined the initial 10 results it asked for in its original request. Whenever the service client requests more results from its local result collection than is currently scheduled to arrive there, the result collection contacts the proxy collection at the server side for more of the hits. These are then delivered just like the original results. Notice that in order to do this, the proxy collection needs to let the client's result collection know the proxy collection's object ID. This is done as one of the parameters in each of the (partial) result deliveries.

2.3 Getting More Properties of a Result Object

In our example, the client asked that each result object contain title and abstract properties. After examining some of the result objects, the client might want to see some more properties of one of the result objects. This might be the publication date, the journal in which the publication appeared, or the full content of the document.

This is done simply by asking the result object for the desired additional properties. The result object experiences a 'property fault' and contacts the proxy collection for the additional information.

2.4 Freeing Proxy Collection Resources

The client may take a long time to ask its local result collection for more result objects, or for more properties of the result objects already partially retrieved. What if the proxy collection does not want to keep the resources associated with a given query indefinitely? It may want to use these resources for other requests, or the external service could be a for-pay service which is very expensive to stay connected to. The DLIOP allows the proxy collection to discard its resources at any time, although of course the intent is to have pending requests serviced.

Allowing proxy collections to shut down the resources associated with a particular query raises two issues: What does a result object at the client result collection do if it is asked for properties it did not yet pull over from

the proxy collection? And what does the client's result collection do if the client asks it for more hits of the same query? The first issue is handled by what we call *access capabilities*. Each result object contains an access capability which contains the information needed to obtain values for the object's properties. The access capability for each result object is passed as a parameter from the proxy collection to the client's result collection when results are delivered. An access capability in turn may contain multiple *access options* which each represent one way of getting the property values. When a result object needs to retrieve additional properties for itself, it initially tries to request them through the first access option in its access capability. If that fails, it tries the next one, and so on. Refer to Figure 3 for an informal illustration of access capabilities. Each access option

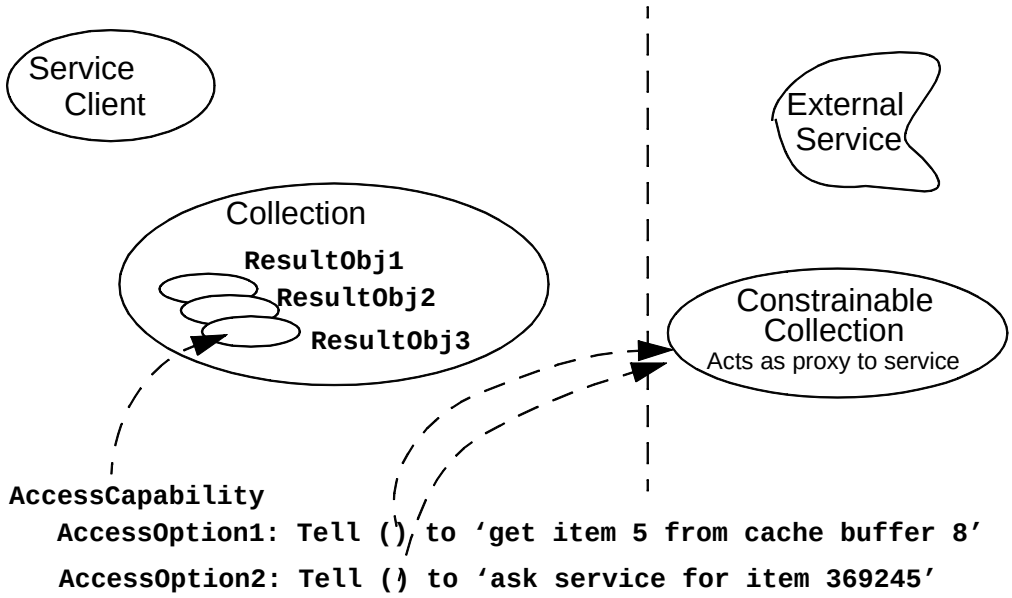


Figure 3: Access Capabilities Provide Flexibility for Resource Management

contains the object identifyer of an object that can provide the additional properties, and a cookie to pass along with the request when contacting that object[1]. For example, the first access option may contain the OID of the proxy collection. The associated cookie tells the proxy collection the ID of the result set it is maintaining within its memory and which it will get the

1. A cookie is a data structure that is passed uninterpreted to its final destination. Only that final destination knows how to interpret and use the data structure.

requested property values from. If the proxy collection has already discarded this result set, it will raise an error in response to the request for property values. The client's result object then attempts this operation again, this time with the second access option of its access capability. The OID may again be the proxy collection. But now the cooky will contain an indication to the proxy collection that a new query is to be performed, with the required information being retrieved (again) from the external service. This is, of course, more expensive than if the first access option had worked, but it allows the DLIOP to avoid sessions which tie up the server side indefinitely.

The second issue, of client collections being asked for more hits after the proxy collection has discarded its resources associated with the query is solved similarly: Every time new results are added to the client's collection, a 'moreCookie' is passed along. Analogously to access capabilities for individual result objects, this data structure provides the client collection with one or more contacts for requesting additional hits. The first will generally be the proxy collection's OID with a cookie that contains the proxy collection's cache pointer to its cached hits, or a handle for getting more information from the external service connection that is kept open. A second contact option could again be the OID of the proxy collection. But this time the cookie would contain instructions for the proxy collection to repeat the entire query.

2.5 Load Balancing at the Server Side

Sometimes it may be desirable to free the object that acts as proxy to the external service for accepting new requests from other clients even before one request's query has been processed to completion. Figure 4 shows how this requirement can be accommodated.

Instead of interacting with the external service itself, the constrainable collection immediately creates a 'delegate' object which takes over the remaining processing of the query request. It then returns to accepting more requests. Note that the request processor object created as the delegate may be created on a machine other than the one running the constrainable collection. The client is unaware of the origin of calls in steps 6 and 7.

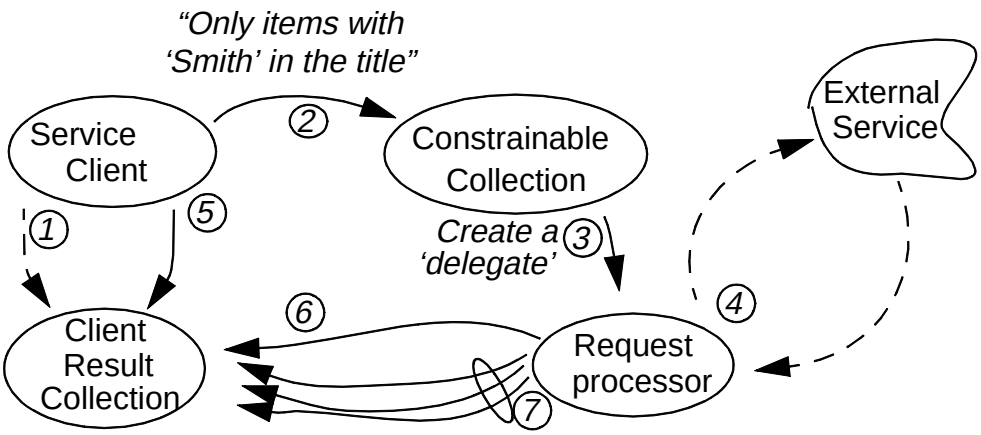


Figure 4: A Search Interaction with Server-side Off-Loading (Interactions 1, 2, 4, 5, 6, and 7 are as in Figure 2)

2.6 Load Balancing During Result Delivery

The DLIOP allows server-side objects to perform load balancing even while results are being delivered to the client. This is easy because, as pointed out in Section 2.2, the OID of the proxy collection is passed to the client's result collection every time a set of results are delivered. Before delegating further work to a new object, the proxy collection can issue a call to the client's result collection, adding an empty set of results, and specifying a new OID as the target for future requests for additional result hits.

3 Technical Details

In this section we describe each of the methods involved in the DLIOP. We use as an example the proxy for Knight-Ridder's Dialog Information Service. In all cases of doubt, the interface specification file available at the testbed Web page reachable from http://www-diglib.stanford.edu has final authority.

3.1 Syntax Conventions

The following examples illustrate the syntax used to describe the arguments to methods and return results. We use this convention to be language-independent. There are straight-forward mappings from this convention to the major programming languages.

- `[]` - an empty list (or sequence)
- `[1,2,3]` - a list of integers 1, 2, and 3. Lists may have arbitrary lengths.

- **`[[], “the”, “an”]`** - a list containing an empty list, the words “the” and “an”.
- **`{}`** - an empty record (with no fields)
- **`{aSize : 20}`** - a record with one field “aSize” with a value of 20.
- **`{aSize : 20, aHandles : [“a”, “b”, “c”]}`** - a record with two fields, “aSize” with a value of 20, and “aHandles” with a value of a list of words “a”, “b”, and “c”.
- **`*aResult*`** - denotes an instance object.
- **`*Result*.AddItems(5)`** - Call the AddItems method on *Result* with an integer argument 5.
- In order to make it easier to match the method calls in the pseudo-code examples below with the corresponding method definitions in the IDL interface definition of Section 4, we name the parameters for which we provide values. For example: for a method defined as:

  ```
  PropertyNamesList  GetItemsPropertyNames(in TCookies pCookies)
  ```

  a example call might be written as:

  ```
  GetItemsPropertyNames(pCookies: [‘foo’, ‘bar’])
  ```

3.1.1 Symbol Prefixing

Parameter names, record field names, class names, and other interface elements are prefixed with one-letter tags. This is done to allow determination of a symbol’s type simply by inspection, without needing to know the context in which it appears. .

Table: Symbol prefixing

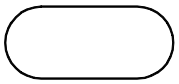
T	Non-class Type	**k**	constant	**I**	Interface Module	**g**	global variable
C	Class Interface	**a**	llocal variable	**E**	Enumeration Type	**p**	parameter to a method
f	instance variable						

By this convention, ‘**`CItem`**’ is a class, ‘**`pItems`**’ is a parameter to a method, ‘**`aItem`**’ is a local variable, and ‘**`fItem`**’ is an instance variable for an object instance.

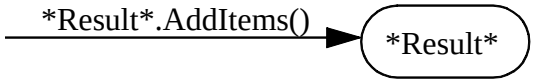
3.1.2 Graphical Conventions

We use the following graphical conventions.

A single process' address space



An instance of a class.



Call the **`AddItems`** method on **`*Result*`**.

3.1.3 Query Format

DLIOP currently uses Z39.50's RPN structure with type 101 format for delivering its queries. This means that the query is delivered fully parsed. See the accompanying IDL file in Section 4 for definitions of the associated data structures, and the Z39.50 documentation for a grammar. Since Z39.50 conventions allow specification of non-101 queries (type 0 queries), clients may choose instead to send query strings or structures of their choice, if the corresponding target objects understand. When using such type 0 queries, the structure used to pass the query itself contains two fields: a query subtype, and a structure holding the query itself, which may just be a string. Within type 0 queries, the DLIOP thus allows callers to identify the particular query language they use. In particular, subtype 0 is the DLIOP front-end query language which InfoBus facilities can translate to a variety of other query languages.

Generating the RPN structure of a type 101 query by reading the IDL specification is not trivial. Conversion code is available in Python and C++ to make this easier.

The property names used in DLIOP are USMARC tags, although in the following we use English names for clarity.

3.1.4 Synchronous vs. Asynchronous Methods

The names of all asynchronous methods begin with **`Request`**, as in **`RequestConstrain`**. These methods return immediately with no result. The protocol describes how these methods subsequently contact the caller to deliver their results.

All methods starting with **`Get`** are synchronous. The result types of sychronous methods are described below.

3.1.5 Stoppping Asynchronous Requests

When it is necessary to stop a running request, the call **`CancelRequest`** will do the trick.

3.1.6 Error Signaling

For synchronous calls, CORBA handles error signalling. The interface file contains a declaration of the errors that may be raised by each method of the protocol. The language binding determines how an error is signalled by the callee, and how it is delivered to the caller. In general, this is done in the way most natural to the particular programming language.

Since asynchronous methods immediately return with no argument, a different mechanism must be used. The DLIOP protocol specifies that each client have the method **`RaiseError`**. It takes a message ID and a record describing the error. The message ID links the error to a previous asynchronous request. If a service encounters an error while servicing an asynchronous request, it calls **`RaiseError`** instead of the regular response method.

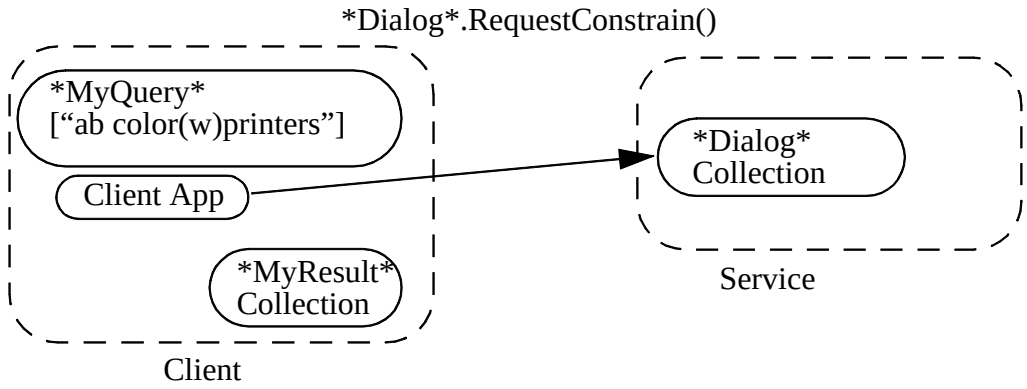
3.1.7 Submitting a Query

A query is delivered by calling method **`RequestConstrain`** on the proxy collection which is labelled **`*Dialog* Collection`** in Figure 5 below. In addition to the object that performs the call, the client side may contain two other objects when the call is made. One is the local result collection object discussed in Section 2.1. The other is a query object.

Query objects may contain many aspects of the query, in addition to the query string itself. This might include processing time limits, or other additional limiters and service instructions not expressed in the query. The

exact properties are not specified. The purpose of the query object is to allow servers to learn more about a query than is passed in the **`Request-Constrain`** call, if appropriate. The information passed in that call is, however, sufficient for most purposes.

The following Figure 5 shows a query being delivered. It shows the request




```
RequestConstrain(
pMessageID : 24601,
pQuerySummary : {
      aQuery : *MyQuery*,
      aQueryDescription : {
            querytype : 0
            query : “ab color(w)printers”}
      aQueryItemProperties : [“Title”, “Author”],
      aMoreSummary : []},
pServicePrefSummary : {
      aServicePreferences: NULL,
      aNumberOfItems: 10,
      aMoreSummary : []},
pResultTarget : *MyResult*)
```

Figure 5: Delivering a query to a proxy collection

for a search on Dialog for documents with the words “color” and “printers” occurring right next to each other in the abstract. Title and author of 10 documents are to be returned to the collection *MyResult*[1].

1. Knight-Ridder’s Dialog service actually contains multiple data files. For simplicity, we assume that this proxy accesses a single such file.

The **`RequestID`** is a message identifier invented by the client. It allows subsequent interactions between the proxy collection and the client to identify the query that stood at the start of the exchange.

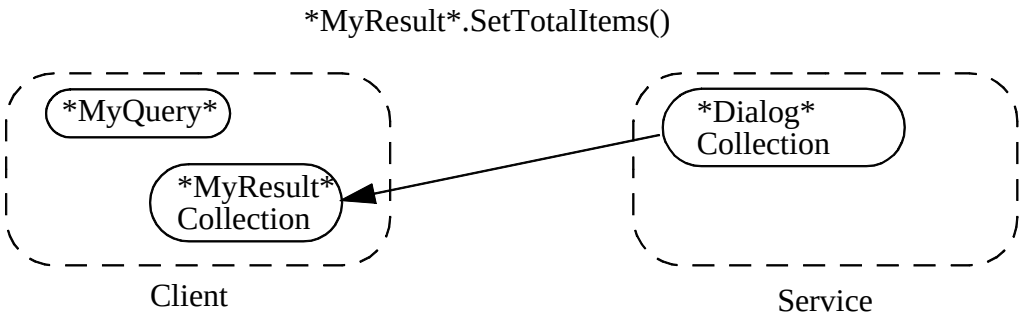
The query summary contains three components: a pointer to the query object (see above), the query, and any other information extracted from the query object to save the proxy collection a call back to the query object. Since in this case we are passing all necessary information in this summary, the query pointer (*MyQuery*) could be set to NULL. In fact, the client is not even obliged to create a query object. The **`aMoreSummary`** summary field is a possibly empty list of property/value pairs.

The service preferences summary part is for communicating general preferences the client might have in dealing with the service. It includes an optional pointer to a preference object which could contain any number of properties the proxy collection might understand. Examples are shortcut definitions, global format preferences, billing, or authentication information. The **`aMoreSummary`** summary field is a possibly empty list of property/value pairs.

The result target is a pointer to the client's result collection where the proxy collection is to deliver the results. Note that while in this exposition we are using separate result collection objects for this purpose, one could write an object which calls **`RequestConstrain`**, and names itself as the recipient of the results. The only requirement is that this object support the client methods explained below.

The search is now performed, and we assume that 59 hits are returned. The client's result collection is informed immediately about the total number of hits to expect (Figure 6). The client can now display the total number of items in the result set collection, even though they are not yet available at the client.

Next, the Dialog proxy collection optionally creates a result collection on its site to handle all subsequent interactions on behalf of this query. It could instead choose to do this work itself. **`*Result1*`** in Figure 7 will request at least the title and author of the first 10 hits from the Dialog service,




```
*MyResult*.SetTotalItems(pTotalSize : 59)
```

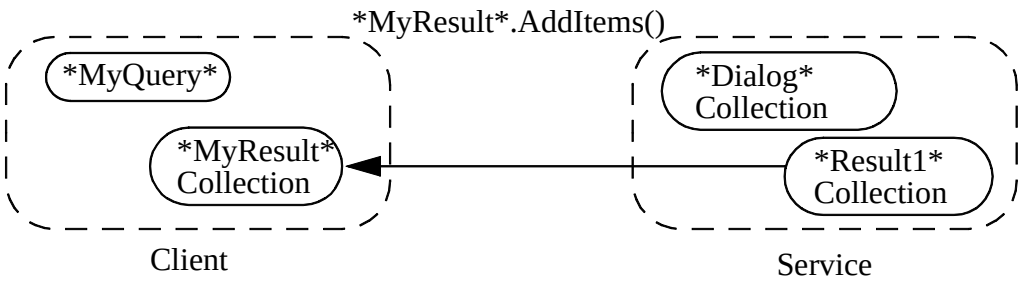
Figure 6: Letting the client collection know the total number of hits found

although it may choose to request more in anticipation of follow-up requests from the client.

Once the titles and authors of at least some of the first 10 hits are available, **`*Result1*`** calls **`AddItems`** on the client's result collection. If retrieval from the external service (i.e. Dialog in our example) is slow, the proxy collection may decide to call **`AddItems`** several times with a few results. Figure 7 shows how the information is delivered to the client. The **`AddItems`** method takes three main parameters: a message ID tying the items being delivered to the previous query they are the results of, properties and access capabilities of the results being delivered, and the contacts for the client to use when requesting additional hits for this query.

The message identifier matches the identifier supplied by the client in its original **`RequestConstrain`** call. If the client's result collection merges the results of multiple queries, this ID allows it to keep track of progress for each query.

The second parameter, the core of the information, is partitioned into two parts: one access capability for each of the 10 hits, and the requested property values (title and author) for each hit. As explained in Section 2.4, an access capability may contain more than one access option. In Figure 7 we see an example. Both access capabilities contain two access options. The first option in the figure indicates that the kind of object to which the properties belong is "document". The object to request more properties from is **`*Result1*`** which is the proxy collection's result collection work-




```
*MyResult*.AddItems(pMessageID : 24601,
  pItems : {
    aACs: [
    [ {anItemClass : “document”,
       aTarget : *Result1*,
       aCookie : “S1: 1”, aHints: []},
      {anItemClass : “document”,
       aTarget : *Dialog*,
       aCookie : “an=3424601”, aHints: [] } ],
    [ {anItemClass : “document”,
       aTarget : *Result1*,
       aCookie : “S1: 2”,
       aHints: []},
      {anItemClass : “document”,
       aTarget : *Dialog*,
       aCookie : “an=4310642”,
       aHints: [] } ],
    ... 8 more Access Capabilities ... ],
    aState : {
      aNames : [“title”, “author”],
      anItemsState : [
        [“Printer and painter primary colors”, “Smith J”],
        [“Color Printers”, “Todd W”],
        ... 8 more records ...] },
    },
  pNewMoreCookie: [
    [anItemClass : NULL,
     aTarget : *Result1*,
     aCookie : “S1: 11”,
     aHints : []],
    [anItemClass : NULL,
     aTarget : *Dialog*,
     aCookie : “Query(color and printers),#11”,
     aHints : []]
    ])
```

Figure 7: Delivering title and author of the first 10 hits to the client

horse. The cookie to pass back with the request for more properties is “S1:1”. This cookie makes sense in the context of Dialog interactions: “S1” refers to a set ID returned by that service. The “1” indicates that the property values come from the first hit in the S1 set. Again, this is a cookie, so it only makes sense to the proxy collection or its workhorse result collection.

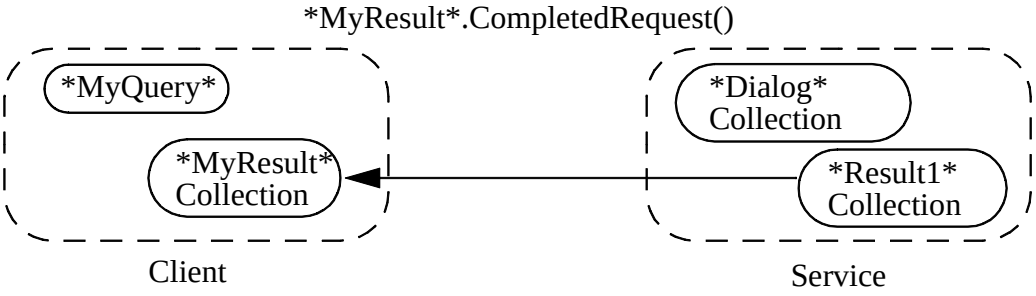
The second access option will be used when the first one fails. This will be the case when **`*Result1*`** has been discarded. Notice that the target now is the **`*Dialog*`** proxy collection itself, and the cookie is an accession number to use when retrieving pieces of this item from Dialog.

The **`aHints`** field can be used for any other necessary information.

The data structure used for transmitting the property values themselves is simple. It contains the list of property names to deliver, followed by a list of lists with the values.

The contact information parameter is similar to the access capabilities. In this case, the first option’s cookie is a pointer into the proxy collection’s cache where the next set of properties to be retrieved is stored, or enough information to get the next set from Dialog itself. The second option contains enough information to re-issue the query.

Finally, in Figure 8, the service informs the client that no more results will be delivered.




```
*MyResult*.CompletedRequest(pMessageID : 24601)
```

Figure 8: Signalling the end of result delivery

Upon receipt of the property values, the client collection creates one object

(of class **CItem**) for each result, and fills the Title and Author properties with the correct values (Figure 9). It remembers the **NewMoreCookie** passed in

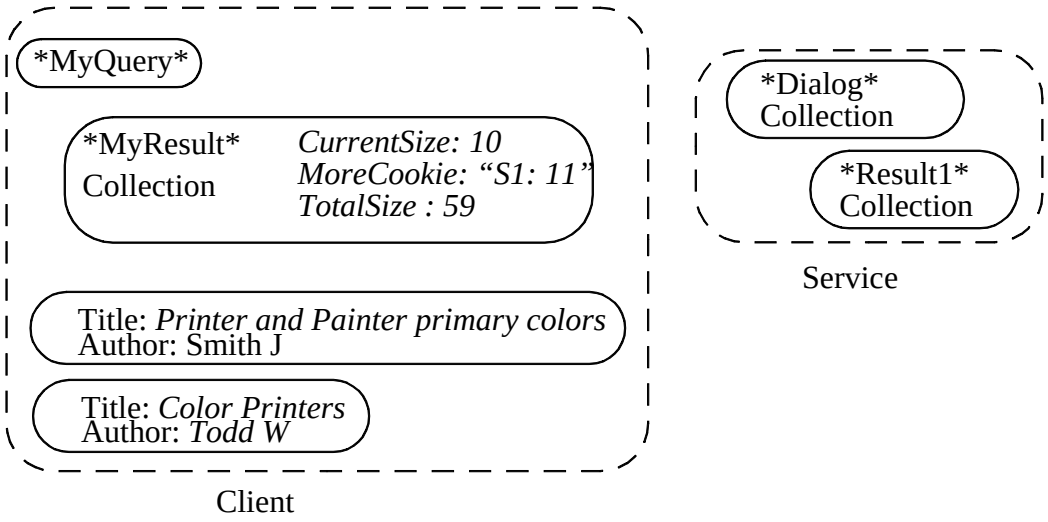


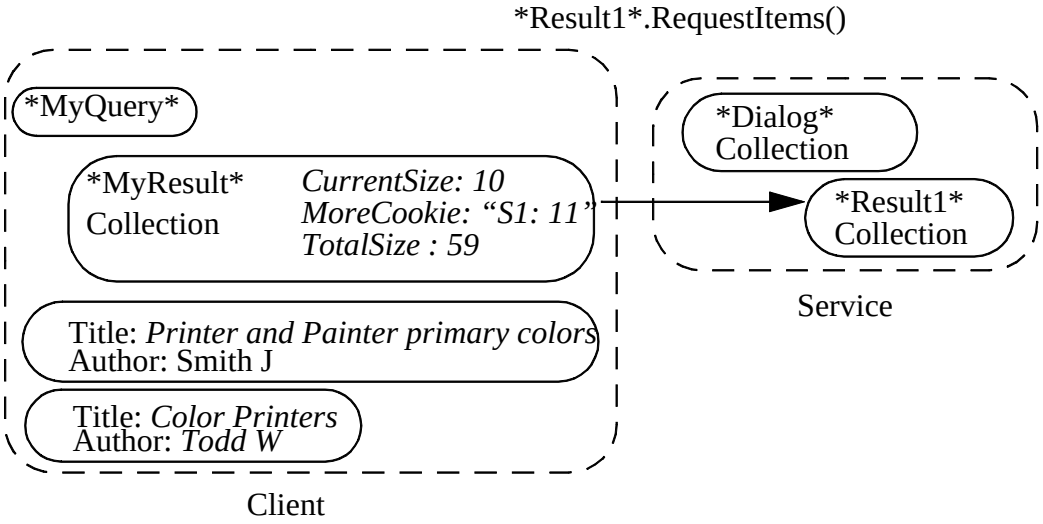
Figure 9: Client collection creates objects and fills them with query results

the **AddItems** call in case it needs to request more results later. Each **CItem** object also remembers its access capability, although these structures are omitted from Figure 9 for clarity.

The client's result collection can now display these result objects. At some point, it may receive a request for some of the additional hits. It then needs to communicate with the service side to obtain the Title and Author properties of those additional results. This is shown in Figure 10. The example shows a request for the titles and authors of 20 additional results. Note that this method call is again asynchronous. Notice also that the client collection uses a different request ID for this call, because the previous request (the original query) has been serviced to completion.

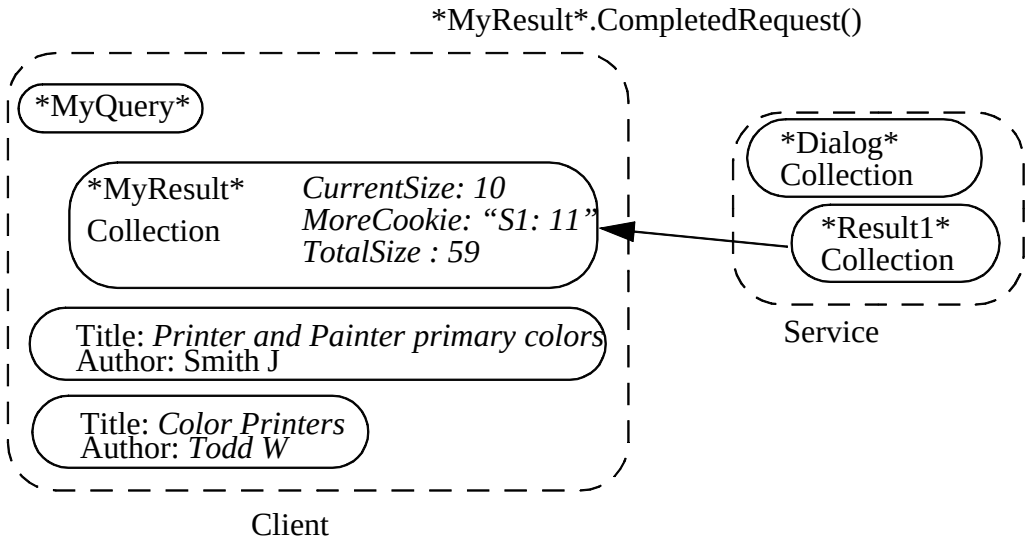
In response, the proxy collection's workhorse requests the 20 additional hits from the external Dialog service (if it did not already cache these in anticipation of this request). The delivery of the title/author property values proceeds just as after the original query request (Figure 11).

After delivery, the **completedRequest** is sent to indicate that all additional results have been delivered (Figure 12)




```
*Result1*.RequestItems(
  pMessageID : 24602,
  pPropertyNames : ["title", "author"],
  pMoreCookie : "S1: 11",
  pNumberItems : 20,
  pResultTarget : *MyResult*
)
```

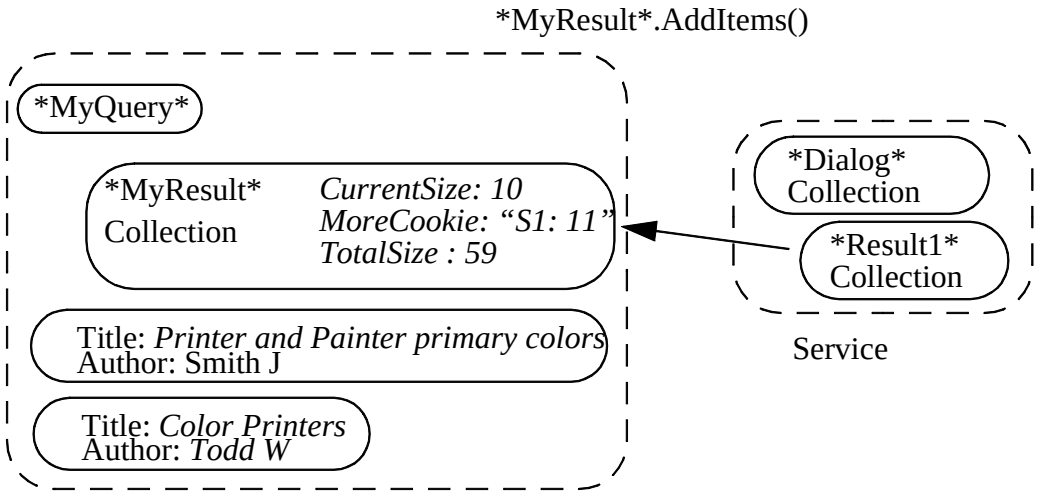
Figure 10: Requesting property values of additional results




```
*MyResult*.CompletedRequest(pMessageID : 24602)
```

Figure 12: Signal that properties for all additional results have been delivered

Again, as in the first round, the result collection builds **`CItem`** instances to




```
*MyResult*.AddItems(pMessageID : 24602,
  pItems : {
    aACs: [
    [ {anItemClass : "document",
       aTarget : *Result1*,
       aCookie : "S1: 11",
       aHints: []},
      {anItemClass : "document",
       aTarget : *Dialog*,
       aCookie : "an=573638",
       aHints: [] } ],
    ... 19 more Access Capabilities ... ],
    aState : {
      aNames : ["title", "author"],
      anItemsState : [
        ["HP Color Printers", "Farmer D"],
        ... 19 more records ...] },
    },
  pNewMoreCookie: [
    [anItemClass : NULL,
     aTarget : *Result1*,
     aCookie : "S1: 31",
     aHints : []],
    [anItemClass : NULL,
     aTarget : *Dialog*,
     aCookie : "Query(color and printers),#31",
     aHints : []]
    ])
```

Figure 11: Delivering properties of additional results

hold the properties, as shown in Figure 13.

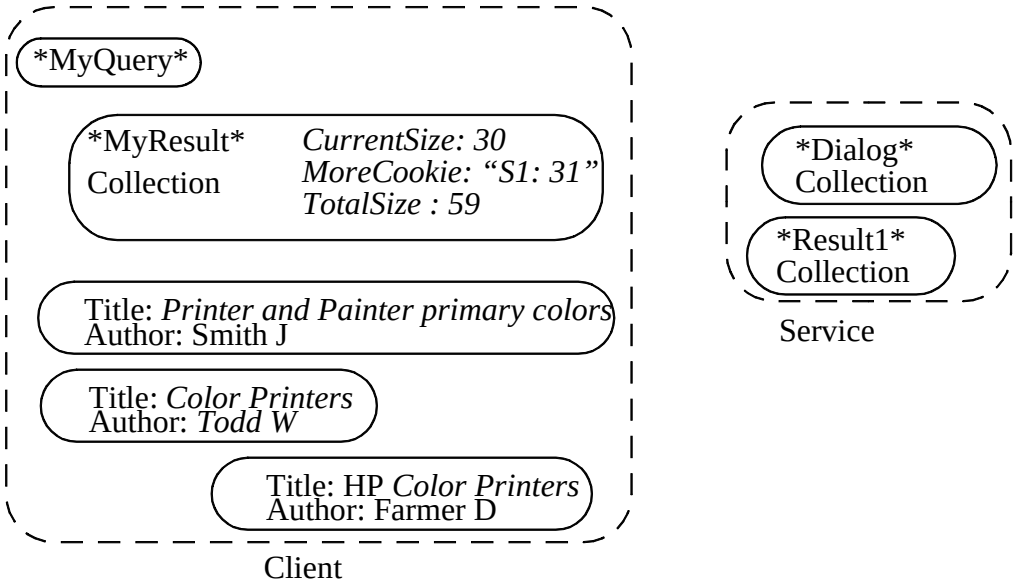
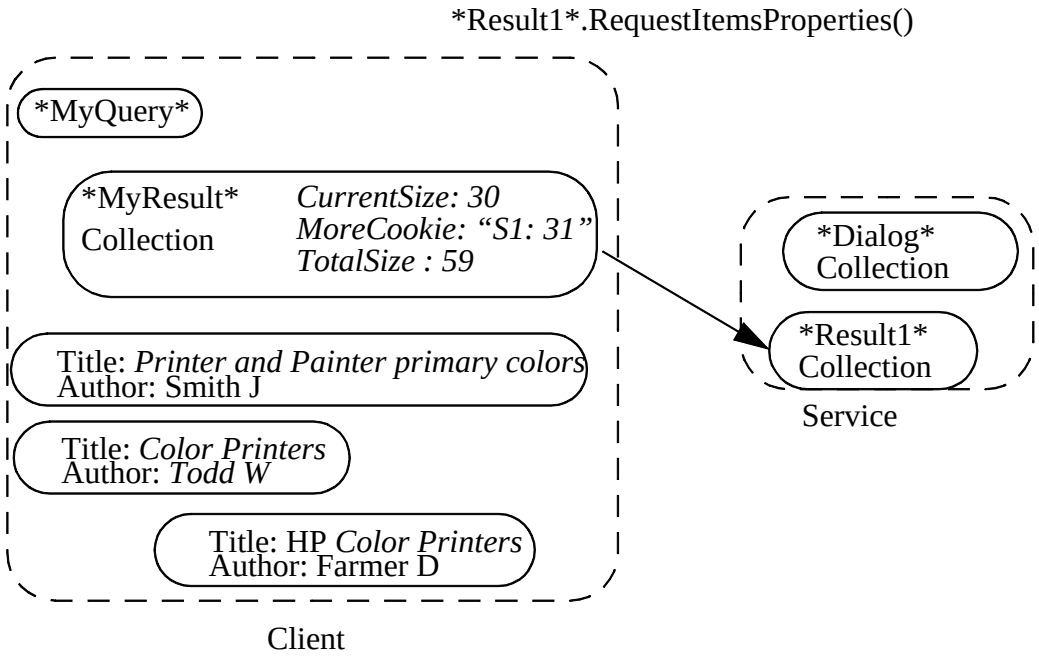


Figure 13: Client materializes objects for additional results

Now, the user requests the abstracts of items 1, 5, and 9. Since these are properties not originally requested, the **`CItem`** instances do not have those property values filled in. The client's result collection can gather all of these property values in one call of the method **`RequestItemsProperties`** on the object it is supposed to ask for additional information. In this case that is the proxy collection's **`*Result1* Collection`**. The request for additional properties is shown in Figure 14. The **`pClientCookie`** parameter will be passed back to the client result collection when the service provides the abstracts. These cookies will then allow the client collection to match incoming abstracts with the correct objects. The client result collection can use any scheme it wishes for these client-side cookies.

As is apparent from the "Request" part of the method name, the **`Request-ItemsProperties`** call is asynchronous. At some later point, after the service has retrieved the requested abstracts, the service needs to deliver them to the client. This happens via the **`SetItemsProperties`** call as shown in Figure 15.

As promised, the client-side cookies (42, 43, and 44) are included as a parameter in the call. In this case, the client only requested the abstract

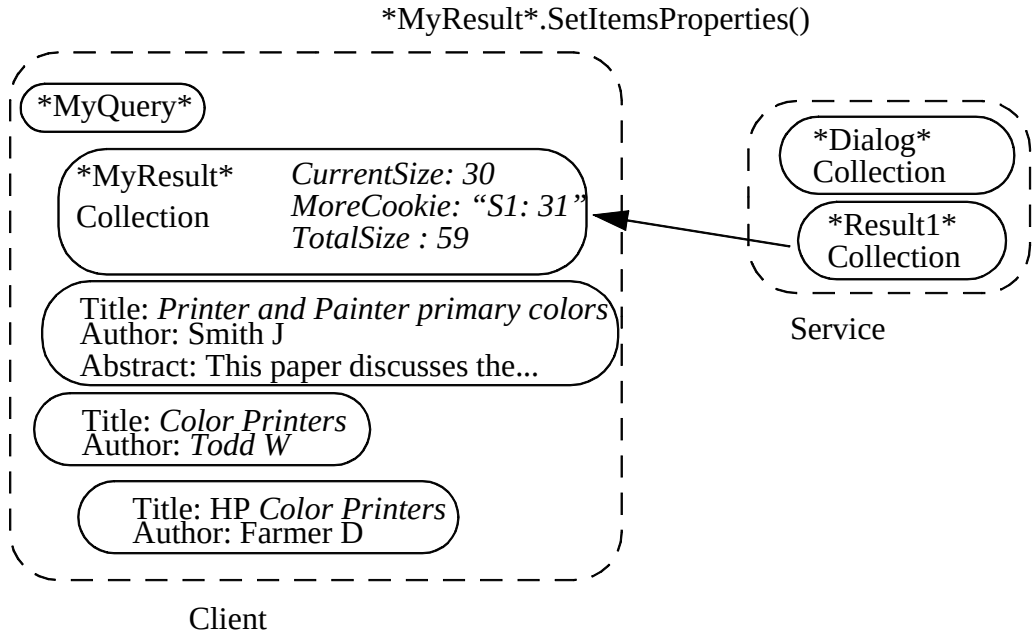



```
*Result1*.RequestItemsProperties(
  pMessageID : 24603,
  pPropertyNames : ["abstract"],
  pServerCookies : ["S1: 1", "S1: 5", "S1: 9"],
  pClientCookies : [42, 43, 44],
  pResultTarget : *MyResult*
)
```

Figure 14: Requesting abstracts for items 1, 5, and 9

property of these three results. It could instead request multiple properties at the same time. In that case, the **aNames** list would contain the names of all the requested properties, and the **anItemsState** parameter would contain a list of lists whose lengths were greater than 1.

As always, the service calls the **CompletedRequest** on the client collection when it is done sending properties (Figure 16). Remember why this is necessary: If it takes a long time to gather all the requested properties for all the results indicated in the **RequestItemsProperties**, the service may decide to call **SetItemsProperties** multiple times with partial results. This ensures that data is moved as quickly as possible. If the client wants to abort work being done on behalf of a request, a **CancelRequest** call may be

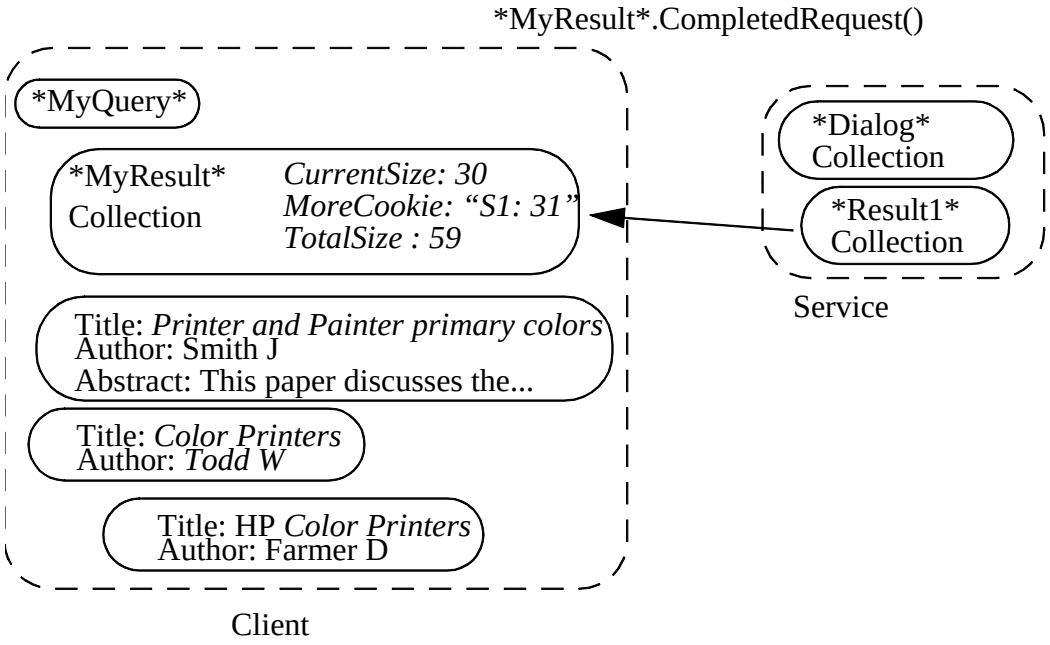



```
*MyResult*.SetItemsProperties(
  pMessageID : 24603,
  pItems : {
    aClientCookies : [42, 43, 44],
    aState : {
      aNames : ["abstract"],
      anItemsState : [
        ["This paper discusses the basics of color
          and the relation to the systematics of painting..."],
        ... 2 more records ...] },
    }
  })
```

Figure 15: Additional abstracts are delivered to client

issued.

In the following appendix, we include the formal CORBA IDL interface definitions for the DLIOP. This includes type definitions for all the data structures used (including the query structures), definitions for all the classes, the signatures of the methods they support, and all exceptions.



`*MyResult*.CompletedRequest(pMessageID : 24603)`

Figure 16: Signal that all abstracts have been delivered

4 Appendix A

Here is an overview of the classes and their methods:

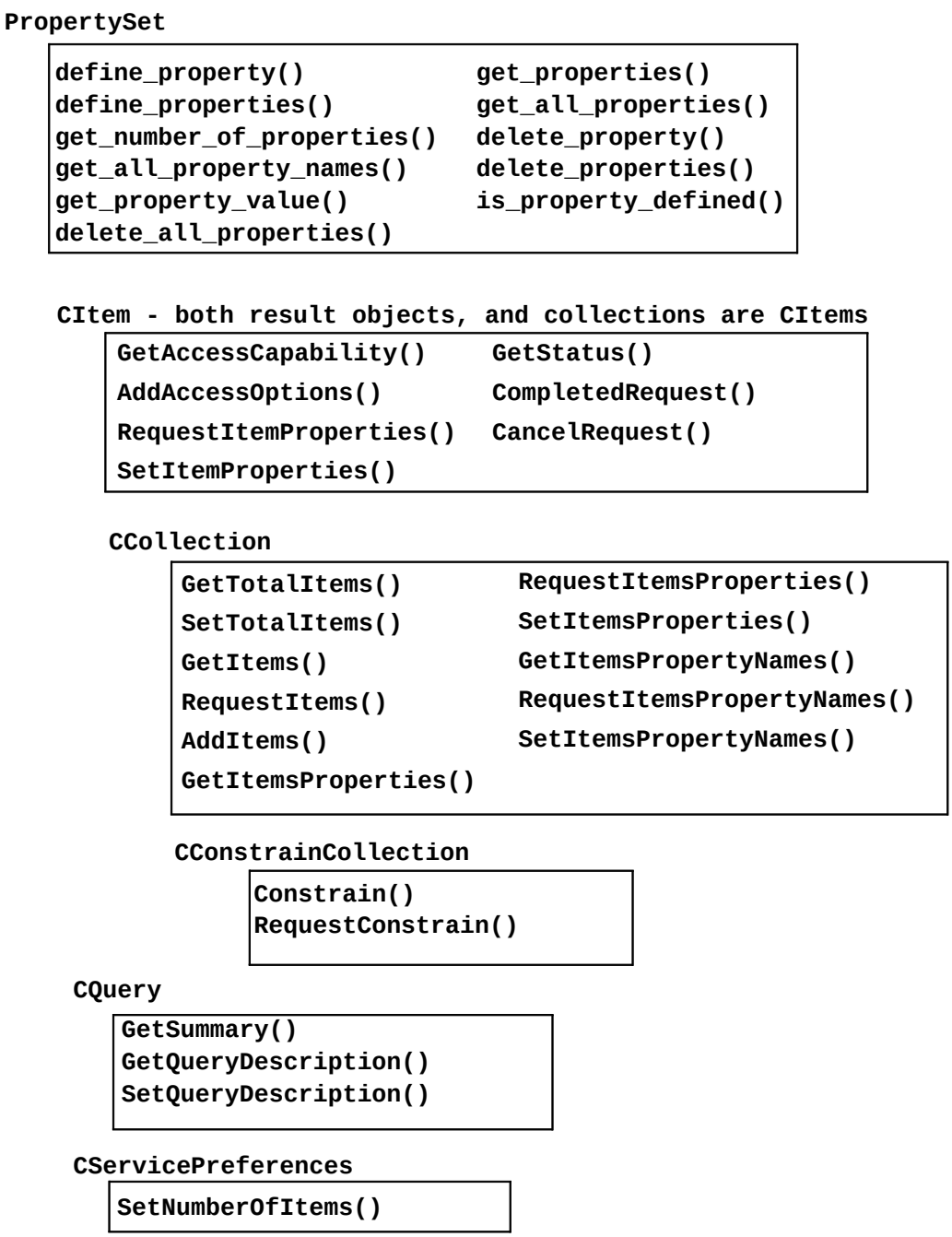


Figure 17: Class Hierarchy of DLIOP Interface
(Indentation indicates inheritance)

```
/* ------------------------------------------------------------ */
/* DLInterchange.idl
/* last update: 9/19/95  hassan */

#include "CosPropertyService.idl"
module IDLInterchange
{

/* ------------------------------------------------------------ */
/* some simple types */

typedef string TString;

typedef unsigned long TOptionalSize;

/* this typedef breaks the generated C++ code */
/* typedef CosPropertyService::Any Any; */

/* Forward references */
interface CItem;
interface CServicePreferences;
interface CQuery;
interface CCollection;
interface CConstrainCollection;

/* ------------------------------------------------------------ */
/* define a cookie. */

typedef CosPropertyService::Any TCookie;
typedef sequence<TCookie> TCookies;

/* ------------------------------------------------------------ */
/* TMessageID is a client-side identifier for all the responses to
   a particular request. */

typedef TCookie TMessageID;

typedef TString TItemClass;
```

```
/* ------------------------------------------------------------ */
/* define Status - for use with CItem::GetStatus call.          */

enum EStatus {
IN_PROGRESS,        /* the request is being worked on. */
UNKNOWN_MESSAGE_ID, /* never heard of it. */
COMPLETED           /* that request has completed. */
};

/* ------------------------------------------------------------ */
/* define Access Capability */

struct TAccessOption
{
    CItem aTarget;
    TCookie aCookie;
    TItemClass anItemClass;
    CosPropertyService::Properties aHints;
};

typedef sequence <TAccessOption> TAccessCapability;
typedef sequence <TAccessCapability> TAccessCapabilities;

/*  ItemState */
/* ------------------------------------------------------------ */
/* define item state */

/* Three types for transferring state for multiple items:
   1. Simplest case: only the state information is transmitted.
   2. Service provides access capabilities and associated state
     for multiple items.  Typical use: Initial response to a query.
   3. Client asked for more properties of items via their access
      capabilities, providing a set of associated client cookies.
     The service now returns the requested properties, each with
    the associated client cookie.

   Note: this trades off interface beauty for minimizing informa
  tion transfer.  :)

*/

/* This is what a CosPropertyService::PropertyValuesList would be
*/
typedef sequence <CosPropertyService::Any> TPropertyValues;
typedef sequence <TPropertyValues> TPropertyValuesList;
```

```
/* swh 9/19/95 */
typedef    sequence<CosPropertyService::PropertyNames>    Proper-
tyNamesList;


struct TItemsState
{
CosPropertyService::PropertyNames aNames;
TPropertyValuesList anItemsState;
};

/* note: aNames and anItemsStates[i] must be of the same length. */

/*  ItemsCookieState */

struct TItemsCookieState
{
TCookies aClientCookies;
TItemsState aStates;
};


/* note: aClientCookies and aStates.anItemsState must be of the
same length. */

/*  ItemsACState */
struct TItemsACState
{
TAccessCapabilities aACs;
TItemsState aState;
};


/* note: aACs and aStates.anItemsState must be of the same length.
*/

/* ------------------------------------------------------------ */
/* Exceptions - some possible exceptions.   Needs work.         */

exception InvalidRequest{TString aReason;};         /* swh 9/19/95
*/
exception InvalidQuery{TString aReason;};           /* swh 9/19/95
*/
exception InvalidAuthenication{TString aReason;};   /* swh 9/19/95
*/
```

```
exception InvalidPayment{TString aReason;};         /* swh 9/19/95
*/
exception UnableToCompleteRequest{TString aReason;};     /* swh 9/
29/95 */
exception      InvalidProperties{CosPropertyService::PropertyNames
aPropertyNames;};
exception InvalidMoreCookie{};
exception InvalidCookies{TCookies aCookies;};
exception InvalidMessageID{TString aReason;};       /* swh 3/21/96
*/

enum EExceptionReason
{
kInvalidRequest,
kInvalidQuery,
kInvalidAuthenication,  /* swh 9/19/95 */
kInvalidPayment,        /* swh 9/19/95 */
kInvalidProperties,
kInvalidMoreCookie,
kInvalidCookies,
kUnableToCompleteRequest, /* swh 9/29/95 */

kInvalidMessageID
};

union TException switch (EExceptionReason)
{
case kInvalidRequest : TString aRequestReason;  /* swh 9/19/95 */
case kInvalidQuery : TString aQueryReason;      /* swh 9/19/95 */
case kInvalidAuthenication : TString aAuthenicationReason;  /* swh
9/19/95 */
case kInvalidPayment : TString aPaymentReason;  /* swh 9/19/95 */
case kInvalidProperties : CosPropertyService::PropertyNames aProp-
erties;
case kInvalidCookies : TCookies aCookies;

case kUnableToCompleteRequest : TString aUnableReason;    /* swh
3.21.96 */
case kInvalidMessageID : TString aMessageIDReason;      /* swh 9/
19/95 */
};

typedef sequence<TException> TExceptions;

exception MultipleExceptions{TExceptions aExceptions;};
```

```
/* ------------------------------------------------------------ */
/* ############################################################ */
/*
   CItem
*/
/* ############################################################ */
/* ------------------------------------------------------------ */

/* ------------------------------------------------------------ */
/* CItem holds all or part of the state of an information object.
*/

interface CItem
 : CosPropertyService::PropertySet
{

    /* return all of the item handles related to this item. */
TAccessCapability GetAccessCapability();

    /* add a list of item handles to this item */
void AddAccessOptions(in TAccessCapability pOptions);

    /* remove this list of item handles from this item. */
void RemoveAccessOptions(in TAccessCapability pOptions);
    /* ------------------------------------------------------ */
   /* RequestItemProperties()

   Request to be sent certain properties for this item.  Sometime
   later, one or more SetItemProperties messages will be sent to
   pResultTarget and finally a CompletedRequest() or
   RaiseException() will be sent to signal the end of the request.
     o pMessageID is the client-side data to pass back in order to
       identify the resulting callback messages.
    o PropertyNames are the list of fields desired.
    o pResultTarget is the target result CItem.

  Note: this simply adds a asynchronous option to the ICosProperty
            interface: get properties from a single item.
    */
```

```
      oneway void
      RequestItemProperties(in TMessageID pMessageID,
                            in CosPropertyService::PropertyNames
                                    pPropertyNames,
                        in CItem pResultTarget);


    /* ------------------------------------------------------ */
    /* SetItemProperties()

   Sets properties of this item.
      o pMessageID is the client-side data that matches the
        pMessageID of the initial request (a
           RequestItemProperties call)
      o pItem contains the state information requested.
    */
      oneway void
    SetItemProperties(in TMessageID pMessageID,
                              in CosPropertyService::Properties
                                    pItem);

    /* ------------------------------------------------------ */
    /* GetStatus()

      returns the status of a particular message request.
    */

    EStatus GetStatus(in TMessageID pMessageID);

    /* ------------------------------------------------------ */
    /* CompletedRequest()

     Signals to the client that the request with pMessageID
     has completed successfully.  No other messages with pMessageID
        shall be sent.
    */

    oneway void
    CompletedRequest(in TMessageID pMessageID);
/* ------------------------------------------------------------ */
    /* RaiseException()

       Signals to the client that the request with pMessageID
       has encountered an exception.  No other messages with
     pMessageID
```

```
       shall be sent.
    */

    oneway void RaiseException(in TMessageID pMessageID,
                               in TException pException);

    /* ------------------------------------------------------- */
    /* CancelRequest()

     Signals to a requestee that the pending request with
     pMessageID shall be cancel.  No other messages with
    pMessageID shall be sent to the requestor.
    */

    void CancelRequest(in TMessageID pMessageID) /* swh 3.21.96 */
       raises(InvalidMessageID);
};

/* ------------------------------------------------------------ */
/* ############################################################ */
/*
   CCollection
*/
/* ############################################################ */
/* ------------------------------------------------------------ */

/* CCollection contains a set of items (maybe documents) and
   efficently allows for the retrieval and setting of the items'
   properties in bulk.

   All of the Request methods are asynchronous in nature.
   The results should come back sometime later with the SetItems
  or SetItemsProperties methods.

   The cookies are generated by the server collection and given
   to the client to hand back when more items are needed.  They
   may be used as an index into an array on the server side.
   Other schemes are allowed. This protocol does not require items
  to be stored in a sequencial manner.
*/
```

```
interface CCollection : CItem
{
    /* ------------------------------------------------------- */
    /* GetTotalItems()

       Returns the total expected size of this collection.
     If NULL, then the expected size is unknown.
    */
TOptionalSize GetTotalItems();

/* ------------------------------------------------------- */
    /* SetTotalItems()

       Sets the total expected size of this collection.
         o pTotalSize is the expected total size of the collection.
           If NULL, then the expected size is unknown.

    */
oneway void SetTotalItems(in TOptionalSize pTotalSize);

/* ------------------------------------------------------- */
    /* GetItems()

    Get 'pNumberOfMoreItems' more Items.
         o pPropertyNames are the list of fields desired.
         o pNumberItems represents the maximum number of Items
           to send back.  If NULL, then return all of the items in
           this collection.
         o pMoreCookie was returned by AddItems that represents
           how to get more Items in the collection.  A value of
           NIL means to start from the beginning of the collection.
         o pNewMoreCookie is a new handle to get more Items.
    */
TItemsACState GetItems(in CosPropertyService::PropertyNames
                       pPropertyNames,
                       in TCookie pMoreCookie,
                       in TOptionalSize
                       pNumberOfItems,
                       out TAccessCapability pNewMoreCookie)
            raises(InvalidRequest,
                InvalidProperties,
                   InvalidAuthenication,
                   InvalidPayment,
                   InvalidMoreCookie,
                MultipleExceptions);
```

```
    /* ------------------------------------------------------ */
    /* RequestItems()

       Request to be sent pNumberItems more Items to pTarget.
       Sometime later, one or more AddItems messages will be
       sent to pResultTarget and finally a CompletedRequest()
       or RaiseException() will be sent to signal the end of
     the request.

       o pPropertyNames are the list of fields desired.
       o pMoreCookie was returned by AddItems that represents
         how to get more Items in the collection. A value of
         NIL means to start from the beginning of the collection.
       o pNumberItems represents the maximum number of Items
         to send back.  If NULL, then return all of the items in
       this collection.
       o pTarget is the result collection to send the Items to.
    */

oneway void RequestItems(
                 in TMessageID pMessageID,
                 in CosPropertyService::PropertyNames
                pPropertyNames,
                 in TCookie pMoreCookie,
                 in TOptionalSize pNumberItems,
                 in CCollection pResultTarget);

    /* ------------------------------------------------------ */
    /* AddItems()

    Add additional Items to this collection.
        o pMessageID is the client-side data that matches the
          pMessageID of the initial request (a RequestConstrain or
           RequestItems call.)
        o pItems contains the state information requested.
        o pNewMoreCookie should be passed to collection to request
          more Items.  A NIL value means there are no more Items.
    */
oneway void AddItems(in TMessageID pMessageID,
                         in TItemsACState pItems,
                          in TAccessCapability pNewMoreCookie);
```

```
/*----------------------------------------- */
    /* GetItemsProperties()

       Get properties of items given their service-side cookies.

        o pPropertyNames are the list of fields desired.
        o pServerCookies is a list of server-side cookies
        of the items requested.

       Note: order and size of result TItemsState must match the
             order and size of the pServerCookies parameter.
    */

TItemsState GetItemsProperties(
            in CosPropertyService::PropertyNames pPropertyNames,
            in TCookies pServerCookies)
            raises(InvalidRequest,
                   InvalidAuthenication,
                   InvalidPayment,
                   InvalidCookies,
                   InvalidProperties,
                   MultipleExceptions);

    /* ------------------------------------------------------ */
    /* RequestItemsProperties()

       Request to be sent properties of items given their
     server-side cookies.

       Sometime later, one or more SetItemsProperties messages
       will be sent to pResultTarget and finally a
     CompletedRequest() or RaiseException()
       will be sent to signal the end of the request.

         o pMessageID is the client-side data to pass back
           in order to identify the resulting callback messages.
         o pPropertyNames are the list of fields desired.
         o pServerCookies is a list of server-side cookies
        of the items requested.
         o pClientCookies is a list of client-side cookies
           to be returned in conjuction with the items'
        state information.  The client can
           place anything in these cookies and is guarateed
         to be matched up with the resulting items.
```

```
         o pResultTarget is the target result collection to
        send the Items to.
    */
oneway void RequestItemsProperties(
                 in TMessageID pMessageID,
                 in  CosPropertyService::PropertyNames  pProper-
tyNames,
                 in TCookies pServerCookies,
                 in TCookies pClientCookies,
                 in CCollection pResultTarget);

    /* ------------------------------------------------------ */
    /* SetItemsProperties()

       Set properties of existing Items in this collection.
       If a item does not exist in this collection, raise an
     exception.
         o pMessageID is the client-side data that matches the
           pMessageID of the initial request
        (a RequestItemsProperties call)
         o pItems contains the state information requested.
    */
oneway void SetItemsProperties(in TMessageID pMessageID,
                    in TItemsCookieState pItems);

    /* ------------------------------------------------------ */
    /* GetItemsPropertyNames()

       For a list of items, return a list of valid property names.

    */

PropertyNamesList
          GetItemsPropertyNames(in TCookies pCookies)
            raises(InvalidRequest,
                   InvalidCookies,
                   InvalidAuthenication,
                   InvalidPayment,
                   MultipleExceptions);
```

```
    /* ------------------------------------------------------ */
    /* RequestItemsPropertyNames()
    */

        oneway void RequestItemsPropertyNames(in TMessageID
                                    pMessageID,
                                              in TCookies
                                    pCookies,
                                   in CCOllection
                                    pTarget);

    /* ------------------------------------------------------ */
    /* SetItemsPropertyNames()
    */

oneway void SetItemsPropertyNames(in TMessageID pMessageID,
                                   in PropertyNamesList
                                     pNamesList);

    /* ------------------------------------------------------ */
    /* RaiseExceptions()

       Signals to the client that the request with pMessageID
      has encountered multiple exceptions.  No other messages with
       pMessageID should be sent.
    */

    oneway void
    RaiseExceptions(in TMessageID pMessageID,
                    in TExceptions pExceptions);

    /* ------------------------------------------------------ */
    /* RemoveItems()

       Remove the following items from the collection.
    */
    void
    RemoveItems(in TCookies pCookies)
        raises(InvalidRequest,
             InvalidAuthenication,
                      MultipleExceptions);

};
```

```
/* ------------------------------------------------------------ */
/* ############################################################ */
/*
   CServicePreferences
*/
/* ############################################################ */
/* ------------------------------------------------------------ */

struct TServicePrefSummary
{
CServicePreferences aServicePreferences;
TOptionalSize aNumberOfItems;
CosPropertyService::Properties aMoreSummary;
};

interface CServicePreferences
:
CosPropertyService::PropertySet
{
TServicePrefSummary GetSummary();
TOptionalSize GetNumberOfItems();
void SetNumberOfItems(in TOptionalSize pNumberOfItems);
};

/* ------------------------------------------------------------ */


/* proximity */
enum RPNNumericRelationType {kRPN_PROX_LESSTHAN,
                             kRPN_PROX_LESSTHANOREQUAL,
                       kRPN_PROX_EQUAL,
                               kRPN_PROX_GREATERTHANOREQUAL,
                       kRPN_PROX_GREATERTHAN,
                       kRPN_PROX_NOTEQUAL};

enum RPNProximityUnitCode {kRPN_CHARACTER,
                     kRPN_WORD, kRPN_SENTENCE,
                     kRPN_PARAGRAPH,
                            kRPN_SECTION, kRPN_CHAPTER,
                           kRPN_DOCUMENT,
                     kRPN_ELEMENT, kELEMENT,
                            kRPN_SUBELEMENT,
                     kRPN_ELEMENTTYPE, kRPN_BYTE};
```

```
struct RPNProximityOperator {
  boolean exclusion;  /* not within x words. */
  short distance;
  boolean ordered;
  RPNNumericRelationType relationType;
  RPNProximityUnitCode unitcode;
};


/* attributes */

struct RPNAttributeElement {
  unsigned long aAttributeSet;
  unsigned long aType;
  unsigned long aValue;
};

typedef sequence<RPNAttributeElement> RPNAttributeList;

/* operators */

enum RPNRelationOperator {kRPN_LESSTHAN,
                          kRPN_LESSTHANOREQUAL,
                    kRPN_EQUAL,
                          kRPN_GREATERTHANOREQUAL,
                    kRPN_GREATERTHAN,
                    kRPN_NOTEQUAL,
                          kRPN_PHONETIC, kRPN_STEM,
                    kRPN_RELEVANCE,
                    kRPN_ALWAYSMATCHES,
                    kRPN_CONTAINS};

enum RPNBooleanOperator {kRPN_AND, kRPN_OR, kRPN_NOT};

enum RPNOperatorKey {kRPN_BOOLEAN, kRPN_PROXIMITY,
                     kRPN_RELATION};

union RPNOperator switch (RPNOperatorKey) {
  case kRPN_BOOLEAN : RPNBooleanOperator bool_oper;
  case kRPN_PROXIMITY : RPNProximityOperator prox;
  case kRPN_RELATION : RPNRelationOperator rel_oper;
};
```

```
/* term */

enum RPNTermValueKey { kRPN_STRING, kRPN_NUMERIC, kRPN_REAL };  /*
swh 3.21.96 */

union RPNTermValue switch (RPNTermValueKey) {  /* swh 3.21.96 */
  case kRPN_STRING : TString string_term;
  case kRPN_NUMERIC : long int_term;
  case kRPN_REAL : double real_term;
};

struct RPNTerm  {  /* swh 3.21.96 */
  RPNTermValue term;
  RPNAttributeList attributes;
};

/* operand */

enum RPNOperandKey {kRPNTERM, kRPNCOLLECTION, kRPNATTRIBUTE};

union RPNOperand switch (RPNOperandKey){
  case kRPNTERM : RPNTerm term;
  case kRPNCOLLECTION : CCollection coll;
  case kRPNATTRIBUTE : RPNAttributeList attrs;
};

/* rpn structure */

enum RPNStructureKey {kRPNTREE, kRPNLEAF};

union RPNStructure switch (RPNStructureKey) {
  case kRPNLEAF : RPNOperand op;
  case kRPNTREE : struct RPNNode {
                        RPNOperator op;
                        sequence<RPNStructure> operands;
                      } rpnRpnOp;
};

/* rpn query 101 */
struct RPNQuery101 {
  unsigned long attributeSetId;
  RPNStructure rpn;
};
```

```
/* rpn query 10 */
struct RPNQuery0 {
  long querytype;
  CosPropertyService::Any query;
};

enum QueryType {kQTYPE0, kQTYPE101};

union TQueryDescription switch (QueryType) {
  case kQTYPE0 : RPNQuery0 QueryType0;
  case kQTYPE101 : RPNQuery101 QueryType101;
};

/* ------------------------------------------------------------ */

/* Query */
struct TQuerySummary
{
CQuery aQueryObject;
TQueryDescription aQueryDescription;
CosPropertyService::PropertyNames aItemPropertyNames;
CosPropertyService::Properties aMoreSummary;
};

/* ------------------------------------------------------------ */
/* CQuery represents the type, keywords, and other data needed to
   perform a constraint on a collection.
*/
interface CQuery
:
CosPropertyService::PropertySet
{
TQuerySummary GetSummary();

TQueryDescription GetQueryDescription();
void SetQueryDescription(in TQueryDescription pQueryDescription);

/* if this inherits from PropertySet, are these two necessary? --
dlk*/
CosPropertyService::PropertyNames GetItemPropertyNames();
void   SetItemPropertyNames(in   CosPropertyService::PropertyNames
pItemPropertyNames);
};
```

```
/* ------------------------------------------------------------ */
/* ############################################################ */
/*
   CConstrainCollection
*/
/* ############################################################ */
/* ------------------------------------------------------------ */

/* ------------------------------------------------------------ */
/* CConstrainCollection is a constrainable collection meaning that
   constraints can be placed on it (like a query or search.)  It is
   up to the implementation whether it wants to create a new col-
lection
    to constrain or constrain this collection even further.  The
pMoreCookie
   is returned to allow for both of these implementations.
*/

interface CConstrainCollection
:
CCollection
{
    /* ------------------------------------------------------ */
    /* Constrain()

       Constrain Collection with pDesignator and return
       pNumberOfItems Items back with properties of
     pPropertyNames.

       o pQuerySummary is the constraints to place on this
       collection.
       o pServicePrefSummary are miscellaneous preferences
         like quality of service, passwords, flow control, number
       of items to send now.
       o pTotalSize is the total expected size of this
         collection. If NULL, then the total size is unknown at
       this time.
       o pMoreCookie should be passed to collection to request
         more Items.  A NIL value means there are no more Items.
    */
```

```
TItemsACState Constrain(in TQuerySummary pQuerySummary,
                  in TServicePrefSummary pServicePrefSummary,
                  out TOptionalSize pTotalSize,
                  out TAccessCapability pMoreCookie)
        raises (
          InvalidRequest,
          InvalidQuery,
          InvalidAuthenication,
  InvalidPayment,
          MultipleExceptions);

    /* ------------------------------------------------------- */
    /* RequestConstrain()

       Request to constrain Collection with pDesignator and
     send results to pTarget.

       Sometime later, one or more AddItems messages will be
       sent to pResultTarget and finally a CompletedRequest()
       or RaiseException() will be sent to signal the end
     of the request.

      o pMessageID is the client-side data to pass back in order
       to identify the resulting callback messages.
      o pQuerySummary is the constraints to place on this
      collection.
      o pServicePrefSummary are miscellaneous preferences
        like quality of service, passwords, flow control, number
      of itmes to send now.
      o pPropertyNames are the list of fields desired.
      o pTarget is the result collection to send the Items to.

    */
oneway void RequestConstrain(
                  in TMessageID pMessageID,
                  in TQuerySummary pQuerySummary,
                  in TServicePrefSummary pServicePrefSummary,
                  in CCollection pResultTarget);

};
};
```