Proposal for I**3 Client Server Protocol

Hector Garcia-Molina, Andreas Paepcke
September 2, 1996
{hector,paepcke}@cs.stanford.edu
http://www-db.stanford.edu

1 Introduction

This document proposes a Corba-based protocol for submitting queries to servers and for obtaining the results. Figure 1 below illustrates how this protocol would be used. On the left is a client process running some application; in Corba terminology, it is called the client object. On the right is the server object. The client submits a query by invoking a method offered by the server. This is an asynchronous call, so the client does not wait. Later on, after the server has obtained answers, it invokes a method on the client. The parameters of this call include the answers. Not all answers have to be returned in a single call to the client. The server can return a partial set of answers, and later on make additional method calls on the client to return additional answers.

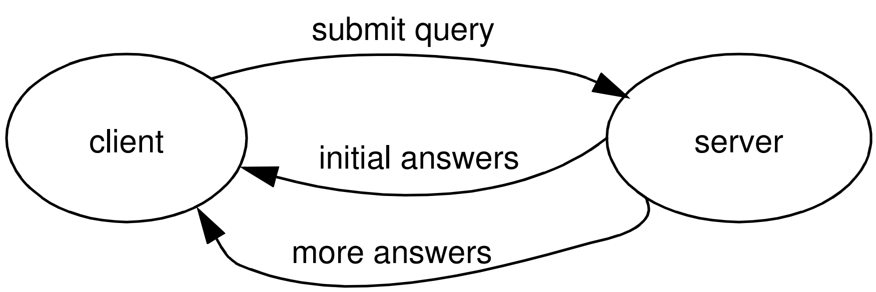

Figure 1: Basics of Protocol

The protocol we propose here is a subset of the Stanford Digital Library Interoperability protocol (DLIOP), which is in use by most of the participants of the Digital Libraries Initiative (DLI) for interacting with other sites. The advantages of adopting a subset of the DLIOP protocol is that we would be compatible with that other DARPA community. Furthermore, the DLIOP protocol has already been tested and found to be effective for incrementally sending answers to clients, something that is also very important in an I**3 environment.

In the rest of this paper we will give examples to show how this protocol works in an I**3 environment. The detailed Corba IDL specification is given in the document “Stanford Digital Library Interoperability Protocol,” available at http://diglib.stanford.edu (under publications). That document also describes additional features of the DLIOP protocol.

2 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 (in the DLIOP document), 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 Submitting a Query

Say the client wishes to submit an MSL query to a server object **`*Server*`**. The following is an example of a call to submit the query.

```
*Server*.RequestConstrain(
pMessageID : 24601,
pQuerySummary : {
      aQuery : NULL,
      aQueryDescription : {
            querytype : 501
            query : "X:-X:<book {<title "cats">}>"},
      aQueryItemProperties:["oid", "label", "type", "svalue", "ovalue"],
      aMoreSummary : [{property_name: "performative",
                        property_value: "do-once"}] },
pServicePrefSummary : {
      aServicePreferences: NULL,
      aNumberOfItems: 10,
      aMoreSummary : [{property_name: "depth",
                        property_value: 2 }]  },
pResultTarget : *Client*)
```

This is an asynchronous call on the server; the client does not wait for the query to be processed. The method is named **`RequestConstrain`** because it constrains the server objects to those that satisfy the query.

The first parameter, **`pMessageID`**, is a transaction identifier invented by the client. It will be used by the server to identify the answers it returns later to the client.

The second parameter, **`pQuerySummary`**, describes the query that is being submitted. Its contents is a record with 4 fields. The first field is not used. (This field is used by the more general DLIOP protocol to send additional query information. The I*3 subset of the protocol may eventually use this field, so it is desirable to leave it in, both for extensibility and for compatibility with the DLIOP protocol.) The second field is itself a record containing two sub-fields. The **`querytype`** field identifies the query language used. We propose to use integers in the range 500 to 599 for I*3 languages; in the example we have assumed that MSL is identified by 501. (Hopefully we will not have anywhere near 100 I*3 query languages!!) The second sub-field, **`query`**, is the actual query string being submitted.The third field is a list of the "properties" that we wish to extract from the matching answers. Since the desired answers are OEM objects, we request to get all their properties, i.e., their **`"oid", "label", "type", "svalue", "ovalue"`**. (The first three of these properties are the normal OEM components. The last two represent the "value" property of the OEM object: if it is an atomic string, then **`svalue`** contains it; otherwise **`ovalue`** contains it. See next section for an example.) The fourth field of the **`pQuerySummary`** record is the

aMoreSummary. This is a list of optional attribute value pairs that give additional requirements for the query. In our sample call we have included the pair “performative: do-once” indicating that we want this query executed once (e.g., as opposed to once every hour). (The default performative is “do-once,” so we actually did not need to include it in our example.) Also notice that we could have more than one pair in the **aMoreSummary**, but for now we only have one possibility defined, the performative. In Section 12 we describe other possible performative values.

The third parameter of the **RequestConstrain** call is the **pServicePrefSummary**. It describes “execution preferences” for the query. Its value is a record with three fields; the first one is not used for now. This second field, **aNumberOfItems**, states the number of answer objects that are desired initially. That is, each query answer is viewed as a set of answer objects. Since there may be many such objects, we can specify how many we wish to retrieve initially. We can later on ask for more, as we will see later. The third field, **aMoreSummary**, optionally gives additional requirements. In our example, we have included the pair “depth: 2” to indicate that we only want the fist two levels of each answer object. (We can use “depth: -1” to indicate we want all levels.) Zero, one, or more service preferences can be listed here. In the future we will define the additional labels that may be allowed here.

The final parameter gives the Corba object that should be called for delivery of the answer. For our example we assume that the ***Client*** itself wants the answer. (In general, the client may ask that some other object get the answers. This gives us a lot of flexibility, for example, when we submit continuous queries.)

4 Getting Initial Answers

Let us assume that the answer to our sample query is the OEM object:

```
<answer {
     <book {
          <title "cats">
          <author "Joe">
          <chapters {
               <chapter "some text">
               <chapter "more text"> } }
     <book {
          <title "cats">
          <author "Fred"> }
      ...rest of books...  }
```

This answer is a single OEM object, but for our protocol we will view it as a set of **book** objects that must be returned. Our query requested 10 of these book objects initially, but the server is free to return a fraction of those in the initial call to the client. Subsequent calls may then return additional results. For our example, let us assume that initially the server only returns the first two book objects. It does this by invoking the following method. (The **chapters** subobject is truncated in this call as will be explained below.)

```
*Client*.AddItems(pMessageID : 24601,
  pItems : {
    aACs: [  ],
    aState : {
      aNames : ["oid", "label", "type", "svalue", "ovalue"],
      anItemsState:
        [["oid-id-1", "book", set, "",
            [["", "title", str, "cats", NULL],
             ["", "author", str, "Joe", NULL],
             ["oid-id-3", "chapters", xset, "", NULL] ],
        ["oid-id-2", "book", set, "", oval:
            [["", "title", str, "cats", NULL],
             ["", "author", str, "Fred", NULL] ]
        ]]  } }
  pNewMoreCookie: [
    [anItemClass : "",
     aTarget : *Server*,
     aCookie : "session-id-55",
     aHints : []]  ]   )
```

The first parameter, **pMessageID**, is the same identifier that was submitted with the query. The second parameter, **pItems**, contains the answer objects. It is a record with two fields. The first field, **aACs**, is not used. The second field actually contains the answers. It is composed of two sub-fields: **aNames** gives the "properties" of the desired retrieved objects. As described earlier, these are the components of OEM objects, i.e., their oid, label, type, and value. (The value is represented by two properties, as explained below.) The second sub-field, **anItemsState**, gives the desired values. It is a list of top-level OEM objects. Each top-level OEM object is itself a list of five values, the first one being the oid, the second one is the label, the third the object type, and the fourth and fifth constitute the value. If the OEM object has an atomic string value, then it is given by the fourth entry; if it is a set value, then it is given by the fifth list entry. Notice that the type of the first four list entries is "string"; the last list entry is a list of lists, just like the original **anItemsState**. (This type is **TPropertyValuesList** (or NULL) in the IDL specification.)

In some cases an OEM object will be "truncated" because its children are at a level that is lower than what the query requested. In this case, the type of this OEM object is reported as **xset**. The **chapters** subobject of the first book retrieved in our example illustrates this. If an atomic string value is too large, the server may send an empty string and give it the type **xstr**. If the client wishes to retrieve the complete OEM object, it can submit a different query, requesting the OEM object, using the provided oid. (If no oid is given, it may have to request an ancestor of the truncated object.) Section 9 explains how to request a truncated object.

Finally, the **pNewMoreCookie** parameter tells the client how to request additional answer objects from the server. Its value is a list of records, each representing one way to access additional answer objects. In our example, we only give one way to access: we can present the Corba object **aTarget: *Server*** with the cookie **"session-id-55"**. (The **anItemClass** and **aHints** fields are not used.) The additional answer objects can be requested via the **RequestItems** method described in Section 8.

5 Total Number of Answers

The server can give the client the total number of answer objects matched by the query. It does this with the call:

```
*Client*.SetTotalItems(pMessageID : 24601, pTotalSize : 59)
```

This value represents the total number of objects, not just those returned by the first **AddItems** call. Notice that the **SetTotalItems** call can be made either before or after **AddItems** is called. As a matter of fact, it is not necessary for the server to call **SetTotalItems**.

6 Sending Additional Answers

The server can send additional answer objects by repeatedly invoking method **AddItems** on the client. The parameters are used in the same fashion as the initial call.

7 No More Answers

The server can tell the client that it has sent all the requested answer objects for a query by calling:

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

Note that this does not mean that all possible answers have been given to the client, only that those requested in the service preferences have been delivered.

8 Requesting Additional Answer Objects

If the client desires more answer objects than those initially requested, it can request them with the following call:

```
*Server*.RequestItems(
  pMessageID : 24601,
  pPropertyNames : ["oid", "label", "type", "svalue", "ovalue"],
  pMoreCookie : "session-id-55",
  pNumberItems : 20,
```

`pResultTarget : *Client*`

The cookie `"session-id-55"` was given in the earlier `AddItems` call. Similarly, the identity of the Corba object to contact, `*Server*`, was given earlier. Notice that this call is also asynchronous. Also notice that the client used the same `pMessageID` as before since this interaction is still part of the original query. This call requests an additional `pNumberItems=20` answer objects. These answer objects are returned via additional calls to the `*Client*`'s `AddItems` method. When these requested 20 objects are returned, the server can invoke the client's `CompletedRequest` method to indicate the interaction is complete.

Note that other service preferences (e.g., depth of answer objects) set in the original query still hold for the additional answers.

9 Fetching Missing OEM Subobjects

As explained earlier, the server can send incomplete OEM objects. If the client wishes to fetch these incomplete objects, it submits a new query. In our example, the first book retrieved was incomplete. The client can formulate the new MSL query `"X:-X:<oem-id-1 book Y>"` to fetch the entire book. Alternatively, it can formulate the query `"X:-X:<oem-id-3 book Y>"` to retrieve only the missing `"chapters"` subobject of the incomplete book. Each of these queries is submitted as described above, giving a larger depth value so that more levels of the book are retrieved. (Recall that "depth: -1" will return the complete answer object.)

10 Errors

If the server encounters an error while processing a client request, it invokes the client method:

`*Client*.RaiseError(pMessageID: 24601, pDescription: "out of money")`

The `pDescription` parameter is a string that describes the error condition.

11 Cancelling a Request

A client can cancel a request by invoking the method

`*Server*.CancelRequest(pMessageID: 24601).`

12 Continuous Queries

There are three basic options for continuous or standing queries: triggered, periodic, and polling. We describe each in turn.

12.1 Triggered Queries

A client can submit a continuous query by including the performative

`{property_name: "performative", property_vale: "triggered"}`

in the `aMoreSummary` field of the `pQuerySummary` parameter of the `RequestConstrain` method. (For simplicity we refer to this performative as "performative: triggered" in this paper.) If this type of execution is requested, the server performs the query once upon receipt, and then continues to evaluate it over new insertions to its collection.

For example suppose that initially a given query retrieves OEM objects *a, b, c*. The server sends them to the client in one or two or three `AddItems` calls. Later on, suppose that the source collection is updated, and that now the OEM objects *a, c, d* satisfy the original query. The server will then send object *d* via another `AddItems` call (with the same `pMessageID` identifier). Note that it will not tell the client that object *b* no longer matches the query. The server continues sending matching objects indefinitely, until the client stops the process with a `CancelRequest` call.

The server can make `SetTotalItems` calls to the client; however, with continuous queries the reported number of answers refers to the number *known so far*. In the example of the previous paragraph, when the server initially evaluates the query and discovers three answer objects, it can make the call

`*Client*.SetTotalItems(pMessageID: 24601, pTotalSize : 3).`

When the additional matching object *d* is discovered, the server can make another call

`*Client*.SetTotalItems(pMessageID: 24601, pTotalSize : 4).`

It is also important to notice that the client must be ready to accept the new triggered answers. For example, say the client initially requests 10 answers, and the initial query evaluation yields exactly 10 answers. The server sends these 10 answers and reports (via a `CompletedRequest` call) that it has delivered all requested answers. If the client wants to receive future new objects, it must make a `RequestItems` call to let the server know it is ready for more answers. On the other hand, if the client requested 10 answers, but the initial evaluation only yielded 7, then the client does not need to send a `RequestItems` call (at least, it will receive the first three new answers).

12.2 Periodic Queries

A variation on continuous queries is to request that changes be detected periodically. For this option, we use the performative `{performative: "periodic"}` in the `aMoreSummary` field of the `pQuerySummary` parameter of the `RequestConstrain` method. We also specify the period in the services preferences section. For instance, if we want changes detected every 90 minutes, we add `{period: 90}` to the `aMoreSummary` field of the `pServicePrefSummary` parameter of the `RequestConstrain` method. (The default is 60 minutes.) Also note that answers in the following period will have the same `pMessageID` (in the `AddItems` calls) as the answers of the previous period.

With this option the client may miss answers that temporarily appear and then disappear. For example, say that initially, objects *a, b, c* match the query and are sent to the client. Thirty minutes later the source is updated and objects *a, b, c, d, e* match. Forty-five minutes later, a deletion occurs, and now only objects *a, b, e* match. Ninety minutes after the query was submitted, the server reports a new answer object, *e.* Thus, object *d* was never seen by the client.

As with triggered queries, the server can report the number of answers known so far. In the example of the previous paragraph, initially it would report 3 answers. Ninety minutes after the initial query, it can report 4 known answers. And as before, after the client receives a `CompletedRequest` call, it must let the server know it is ready for more via a `RequestItems` call. In particular, even if the client has received all the known answers, it must ask for more so it can be ready for the answers of the next period.

12.3 Polling Queries

The third option for continuous queries is to use "polling." Initially, the client makes a `RequestConstrain` call, including the record `{performative: polling}` in the `aMoreSummary` field of the `pQuerySummary` parameter. The server evaluates the query, reports the number of answers (via a `SetTotalItems` call) and sends all current matching objects. In the last `AddItems` call (the one that gives the last known answer object), it includes an access capability (see Section 4) that the client will use when it wants to poll. At some later point in time, the client makes a `RequestItems` call to the server, giving the cookie of the last `AddItems` (and the `MessageID` of the original query). The server then looks for new matching objects at this time, and sends them to the client via one or more `AddItems` calls. The last one of this batch again includes the access capability to be used the next time the client wants to poll. As with the other types of continuous queries, the client can use the `CancelRequest` call to terminate the continuous query.

To illustrate, say the client sends a query that initially yields 10 answers but it only requests the first 7. The server makes the call

```
*Client*.SetTotalItems(pMessageID: 24601, pTotalSize : 10),
```

and then sends 7 answers with one or more `AddItems` calls. Next the server makes a `CompletedRequest` call. When the client is ready for more initial answers, it makes a `RequestItems` call, asking for 3 more answers. The server sends them, say in a single `AddItems` and makes another `CompletedRequest` call.

The client saves the cookie included in the last `AddItems`, and uses it, say two hours later to poll the server. At that point, it gives the cookie back and requests say 5 additional answers (in a `RequestItems` call). Let us assume that only two new objects are found. The server reports

```
*Client*.SetTotalItems(pMessageID: 24601, pTotalSize : 12),
```

sends the two objects in say a single `AddItems`, and makes a `CompletedRequest` call. At this point, the client had requested 5 objects, but only 2 were found; thus 3 are still "missing." The server ignores these 3 missing objects: it simply assumes that the request had been for only 2 objects. (If we do not make this assumption, the server could then send future matching answer objects without the client asking for them. This is not good because under this option the client is supposed to poll.) The client again saves the cookie in the last `AddItems`, and is ready to repeat the cycle.

In summary, the request for more answers, `RequestItems`, is used slightly differently for polling. With triggered or periodic queries, the client must request more answers in anticipation of future answers, so that the server is free to send them as they are generated. With polling queries, the client does not issue the `RequestItems` for new answers until it wants to poll the server.

13 Summary

In spite of its flexibility, we believe that the protocol we have described is relatively easy to implement. The client only has to support four method calls: `AddItems`, `SetTotalItems`, `CompletedRequest`, and `RaiseError`. The server only needs to support three calls: `RequestConstrain`, `RequestItems`, and `CancelRequest`.

Finally, recall that in this note we have described a subset of the more general DLIOP protocol. The more general protocol includes synchronous calls, load balancing options at run time, and other features. We may wish to include some of these features into the I**3 subset at some later point in time. This would simply involve adding methods to the client and server objects, or using some of the parameters we have left unused in the current methods. It would not involve changing the methods we have described here, which would make older systems “backwards compatible.”