A Comprehensive Model for Arbitrary Result Extraction

Neal Sample, Dorothea Beringer, Gio Wiederhold
Computer Science Department
Stanford University, Stanford CA 94305
1-415-725-8363
{Nsample, Beringer, Gio}@cs.stanford.edu

ABSTRACT

Distributed objects and remote services adhere to various standards for data delivery and result extraction. There are multiple means of requesting results and multiple ways of delivering those results. By examining several popular and idiosyncratic methods, we have developed a comprehensive model that combines the functionality of all component models. This model for arbitrary result extraction from distributed objects provides increased flexibility for object users, and an increased audience for module providers.

Keywords

Distributed objects, remote services, result extraction, autonomy, partial extraction, progressive extraction.

1. INTRODUCTION

1.1 Traditional RPCs and asynchronous extraction

We address the problem of obtaining results from any of multiple computational servers in response to requests made by a client program. The simplest form of result extraction is the traditional synchronous remote procedure call (RPC). Parameters are passed in, the client waits patiently for the results, and finally all results are simultaneously available. Only a single object is returned with certain function calls, but most languages offer procedure calls where you can have more than one OUT-parameter. It is even possible in C++ with the use of pointers, and it is cleanly implemented in CORBA. This has been the dominant form of result extraction in most programming languages and for many distributed systems (e.g. CORBA, where a typical procedure call is defined in the IDL syntax like `void mymethod(in TYPE1 param1, out TYPE2 param2, out TYPE3 param3)`.

1.2 Alternative extraction models

There are other models of data extraction that have received little attention in most programming languages and paradigms. These alternative models generally are strictly more functional than the traditional procedure call, and are being used increasingly as a replacement to RPC-style result extraction. They all, in at least some aspect, provide more functionality or flexibility than the RPC. One of the most important enhancements has been the addition of asynchrony. Clients that do not have to stall while results are computed or delivered are strictly more powerful than those that must. Asynchronous result extraction has been used even in many sequential computing languages like Java and Ada, and is enabled in various ways, through message passing, threading, rendezvous, etc. Asynchrony as a tool to delay or reorder result extraction is well understood and widely used [12, 13].

Partial extraction of data results has become almost ubiquitous with the advent of web browsing. There are examples of other extraction models (such as the progressive extraction of results) that have seen limited use in specialized arenas. We can also envision other models encompassing some of the more esoteric extraction models that are potential more flexible than any in wide or limited use today. Perhaps the most important of these model we refer to is “partial extraction.” Partial extraction is taking only the desired portions of a result set, thus saving the costs associated with extracting the entire set. For instance, almost all modern web browsers have the ability to download only text, without images, to speed browsing on slower connections. Many browsers also allow users to filter unwanted objects (i.e., embedded audio) out of html documents. The web has brought “partial extraction” to the desktop as the default browsing model.

Another model for result extraction, again strictly more powerful than the traditional RPC, is the “progressive extraction” model. In many scientific computations, such as adaptive mesh refinements, answers become “better” (e.g., more precise) over time. It can be very important to extract results as progress is made toward an acceptable solution for steering, early termination, or other reasons. A typical application of this type is the design of an aircraft wing [2, 3, 5, 12]. Certain scientific and mathematical processes, like Newton’s method of successive approximation for roots of an equation, can also utilize this type of extraction [4]. The traditional RPC result extraction model does not lend itself well to progressive extractions.

Decomposition of the traditional call model has been discussed for some time [10]; we advocate a further breakdown of the extraction phase of this model. We propose an extraction model, developed in the course of research and development of the language CLAM (Composition Language for Autonomous

Megamodules) [8] within the CHAIMS (Composing High-level Access Interfaces for Multisite Software) megaprogramming project [1], that encompasses these three important augmentations (asynchronous, partial, progressive) to RPC-style result extraction models. The amalgamation of these three extraction paradigms leads to a general model, more expressive than any of the three taken alone.

1.3 Current system support for partial and progressive extraction

For many languages and systems, the generic asynchronous remote procedure call model of result extraction provides enough flexibility. In cases where it does not, custom solutions abound. Occasionally, these custom solutions become widespread, often circumstantially, e.g., because of runtime support rather than language specification. For instance, partial extraction within the available community of web-browsers does not arise from html, per se. It is a consequence of parsing html pages and making only selective *http* requests. In effect, the web-browser implements the partial extraction of results while the html provides a "schema" of the results available. For example, when web users choose not to download certain types of content, the web-browser implements the filtering and is not affected by the html or the http protocol.

We accept this html/http/browsing model as appropriate for partial extractions when a schema for the data is available. Of course, without such a schema, partial extractions would be meaningless. At some level, this constrains the domain for which partial extraction is semantically meaningful or even practical. However, since the model of partial extraction wholly encompasses the traditional RPC method of result extraction, this is not a problem.

Progressive extractions are frequently consequences of elaborate development projects or again arise as a consequence of the nature of the data involved. We can look again to web-browsing to find (an extremely limited) form of progressive extraction, one that arises from the data at hand rather than html or http. Within certain result sets, like weather maps, the results change over time. By simply re-requesting the same data, progressive updates to the data may be seen by a client. On the other hand, to see a broader picture of progressive result extraction, we turn to (usually) hand-tooled codes specialized for single applications.

As far as we know, there is currently no language with primitives supporting progressive extractions. Additionally, what marginal runtime and protocol support for progressive extractions there is seems to only exist because of specialized data streams (like browsing weather services), and not because of intentional support. We have found examples of hand-coded systems that allow partial result extractions, at predefined process points, to allow early result inspection [3]. Such working codes have been built to allow users to steer executions and to test for convergence on iterative solution generators. However, a language and runtime system to develop arbitrary codes that allow progressive extractions is not to be found. This is especially true of compositions languages geared toward distributed objects.

Herein we outline a set of language primitives expressive enough to capture each of these result extraction models simultaneously. We also review an implementation of this general result extraction model, as encompassed within the megaprogramming language CLAM and the supporting components within the CHAIMS system [7].

2. RESULT EXTRACTION WITHIN CLAM

2.1 Definitions and motivation

We focus on result extraction. In our studies, interesting "results" come from computational services rather than simple data services like databases, web servers, etc. Computational services are those that add value to input data. Results from computational services are tailored to their inputs, unlike data services where results are often already available before and after the service is used (like static pages from a web server). Various extraction methods have potentially more value in the context of computational services than in the context of data services. We define "partial extraction" as *the extraction of a subset of available results*. We define "Progressive extraction" as the *extractions of the same result parameter with different content/data at different points in a computation*. The different points of computation can deliver different accuracy of specific results, as is typical for simulations or complex calculations for images. Or they can signify dependencies of the results on the actual time, as is the case for weather data that changes over time.

We went through several versions of CLAM that led to our current infrastructure. Originally limited to partial data extractions, repeated iterations in the design process yielded the general-purpose client-centric result examination and extraction model that seems to be maximally flexible. We say this model is "client-centric" because, unlike an exception or callback-type model, clients initiate all data inspection and collection. In this "data on demand" approach, clients expect that servers do not "push" either data or status, but rather must make requests for both. We specifically contrast this client-centric approach to the CORBA event model seen in 3.4.

Finally, we present CLAM as one possible language and implementation of our generic result extraction model. CLAM is a composition language, rather than computationally oriented language [8, 14]. As such, it is aptly suited to the *presentation* of this extraction model, though not necessarily a programmer's language of choice for a given problem. We advocate this extraction model here, and present it within a particular working framework (CHAIMS).

2.2 Language specification

In CLAM, there are two basic primitives essential to our result extraction model: EXAMINE and EXTRACT. We use EXAMINE to inspect the progress of a calculation (or method invocation, procedure, etc.) by requesting data status or, when provided by the server, also information about the invocation progress concerning computation time.

EXAMINE

- **Purpose**

The EXAMINE primitive is used to determine the state and progress of the invocation referred to by an ***invocation_handle***. EXAMINE has two status fields: state and progress. The state can be any of {DONE, NOT_DONE, PARTIAL, ERROR}. The progress field is an integer, used to indicate progress of results as well as of the invocation.

The various pieces of status and progress information are only returned when the client requests them, in line with the client-centric approach.

- **Syntax**

```
(mystatus=status)                                        =
invocation_handle.EXAMINE()

(mystatus=status)                                        =
invocation_handle.EXAMINE(parameter)

(mystatus=status,     myprogess=progress)     =
invocation_handle.EXAMINE()

(mystatus=status,     myprogess=progress)     =
invocation_handle.EXAMINE(parameter)
```

Imagine a megamodule (a module providing a service) that has a method "foo" which returns three distinct <results/data elements>, A, B, and C. If *foo* is just invoked, and no work has been done, A, B, and C will all be incomplete. A call to `foo_handle.EXAMINE()` will thus return NOT_DONE. When complete, a call to `foo_handle.EXAMINE()` will return DONE, because the work has been performed. If there are some meaningful results available for extraction before *all* results are ready, `foo_handle.EXAMINE()` returns PARTIAL. In case of error with the invocation of *foo*, ERROR is returned.

If the megamodule supports invocation progress information, (mystatus = status, myprogess = progress) = invocation_handle.EXAMINE() is used instead of (mystatus = status) = invocation_handle.EXAMINE() in order to get progress information about the invocation. This progress information indicates how much progress the computation has made already in terms of time used and estimated time needed to complete. Ideally this progress information is in agreement with pre invocation cost estimation which is also provided by CLAM (see primitive ESTIMATE in [8]). Yet as conditions like server load can change very rapidly, it is essential to be able to track the progress of the computation from its invocation to its completion.

Upon receipt of invocation status "PARTIAL," a client knows that some subset of the results are extractable, though not that any *particular* element of the result data is ready for extraction (nor that the result contains progressive and thus temporary values) or has been finalized. Yet PARTIAL indicates to the client that it would be worth while to inspect individual result parameters to get their particular status information.

Once again, imagine *foo* with return data elements A, B, and C. In this case, A is done, B is partially done with extractable results available, and no progress has been made on C. A call to `foo_handle.EXAMINE()` would return PARTIAL, because some subset of the available data is ready. Subsequently, a client issue of `foo_handle.EXAMINE(A)` will return DONE, `foo_handle.EXAMINE(B)` will return {PARTIAL, 50}, and `foo_handle.EXAMINE(C)` will return NOT_DONE. Interpretations of the results from the examination A and C are obvious. In the case of result B, assuming that the result value is 50% completed, and the server makes this additional information available to the client, a return tuple such as {PARTIAL, 50} would express this.

Remember, the second parameter returned by EXAMINE is an integer. CLAM places no restriction on its general use. Servers impart their own meaning on such parameters. However, the recommended usage is for the value to indicate a "percentage (0-100) value of completion." Such semantic meaning associated with a particular server is available to client builders via a special repository.

CLAM couples the EXAMINE primitive with an equally powerful EXTRACT mechanism.

EXTRACT

- **Purpose**

The EXTRACT call collects the results of an invocation. A subset of all parameters returned by the invocation can be extracted. In fact, parameters which are extracted are the ones explicitly specified at the left hand side of the command.

- **Syntax**

```
(myvar1=outvar1,      myvar2=      ...)      =
invocation_handle.EXTRACT()
```

Returning to our previous example of the method *foo*, we look at EXTRACT. The return fields are name/value pairs, and may contain any subset of the available data. For the example method *foo*, we might do the following: `(tmpa=A, tmpb=B) = foo_handle.EXTRACT()`. This call would return the current values for A and B, leaving C on the server.

This extract primitive allows to extract just those results that are ready as well as needed. This in contrast to a much simpler extract where simply all results would be returned with each extract command, independent of their state.

2.3 Analysis of the extraction model

Different flavors of extraction are available with different levels of functionality in EXTRACT and EXAMINE. We present here the various achievable extraction modes when only portions of the complete set of CLAM data extraction primitives are available. This analysis shows how EXAMINE and EXTRACT relate and interact to provide each of the result extraction flavors outlined in this paper.

Within EXAMINE, simple inspection returns a per invocation status value from {DONE, NOT_DONE, PARTIAL, ERROR}. There are two augmentations made to the EXAMINE primitive. The first augmentation is the addition of a second parameter, the progress indicator. The second augmentation is the ability to inspect individual parameters. These two augmentations provide four distinct possible examination schemes:

(1) *without* parameter inspection, and *without* progress information

(2) *without* parameter inspection, but *with* progress information which is invocation specific

(3) *with* parameter inspection, but *without* progress information

(4) *with* parameter inspection, and *with* progress information which is parameter specific

There are two types of EXTRACT in CLAM:

(a) EXTRACT returns all values from an invocation (like an RPC)

(b) EXTRACT retrieves specific return values (like requesting specific embedded html objects).

The two extraction options, when coupled with the four possible types of EXAMINE, form eight possible models of extraction.

Error! Reference source not found. shows the table of possible combinations and outlines basic functionality achievable with each potential scheme. Even with the simplest case as shown in entry 1a, an EXAMINE that works on a per invocation basis only (cannot examine particular parameters) and an EXTRACT that only returns the entire set of results. The extraction model remains more powerful than a typical C++/Fortran-like procedure call because of the viable asynchrony achieved. The model retains its client-centric orientation. The client polls at chosen times (with EXAMINE) and can extract the data whenever desired. The assumption in CLAM is that the client may also extract the same data multiple times. This places some burdens on the support system that we discuss in the next section.

Table 1. EXAMINE/EXTRACT relationships

	single EXTRACT (a)	**per return element EXTRACT (b)**
per *invocation* EXAMINE, One parameter (1)	1a. Like an asynchronous procedure call **e.g., Java RMI**	1b. **Limited partial extraction**. Like 1a, with the added ability to extract a subset of return data at a time indicated by PARTIAL (limited to one checkpoint) or after all results are completed.
per *invocation* EXAMINE, Two parameters (2)	2a. Very limited. *Progressive* extraction becomes possible, with data completion level indirectly indicated by second parameter. Must retrieve entire data set.	2b. Very limited. Progressive extraction still possible, no legitimate potential for partial extractions other than a unique set as in 1b.
per *result* EXAMINE, One parameter (3)	3a. Allows for data retrieval as particular return values are complete, but entire set must be retrieved each time (***semantic* partial extraction**).	3b. **True *partial* extraction** becomes possible here. No real progressive extraction. **e.g., Web-browsing**
per *result* EXAMINE, Two parameters (4)	4a. ***Progressive* extraction** becomes possible, with data completion level indicated by second parameters. Must retrieve entire data set. Can determine more detail than 2a.	4b. ***Partial* and *progressive*** extraction are both possible. Single results may be extracted at various stages of completion. **e.g., CLAM**

The mechanism of table entry 1b is mainly used for partial result extraction when all results are completed, yet not all results are needed. The client has the possibility to only extract those results needed right away, and can leave the other results on the server until they are extracted at a later point or time, or the client indicates it is no longer interested in the results. The main advantage of this level of partial extraction is the avoidance of transmitting huge amounts of data when it is not needed. This is particularly the case when one result parameter contains some meta-information that tells the client if it needs the other result parameters at all. This mechanism can be compared to the partial download of web-pages by a web-browser. In a first download the browser might exclude images. At a later point the person using the web-browser might request the download of some specific or all images based on the textual information received in the first download. This usage of partial extraction has become very important as it allows to partially download small amount of information, and only spend the costs (mainly time when using slow connections) for the costly images when really needed.

There is a limited capability for process progress inspection even with only one return parameter in table entry 1b. With the set {DONE, NOT_DONE, PARTIAL, ERROR}, there is room for *limited* process inspection. The return of “PARTIAL” from a server may indicate a unique meaningful checkpoint to a client. It could be used to indicate some arbitrary level of completion or that some arbitrary subset of data was currently meaningful. This single return value is really a special binary case of the second return value from EXAMINE. Together with a per return element EXTRACT, this model can only reasonably be used to extract one specific, pre-defined subset of results before final extraction of the entire return data set. Figure 1 one shows this use of PARTIAL for creating a single binary partition of results this way: PARTIAL indicates in this specific case that the results A and B are ready, yet C is not yet ready. If e.g., A and C were ready yet not yet B, this could not be indicated and the status NOT_DONE would be returned.

For clarity, we examine more closely the power associated with each entry in the above table. Entry 1a, when there is only one status parameter per invocation (i.e., for all of *foo*) and only the ability to extract the entire return data set, is clearly the most limited. It is very much like an asynchronous remote procedure call, with the clear distinction that the client polls for result readiness. Also data are only delivered when the client requests it, as noted previously.

In table entry 1b, we see that adding per element extraction capability allows clients to reasonably extract one portion of the data set if it is done earlier than the whole. This capability is still very limited. This model can only reasonably be used to extract one specific, pre-defined subset of results before final extraction of the entire return data set. The return value "PARTIAL" can be used to indicate that a *partial* set of results are done, whereas the examination return value "DONE" may indicate that *all* results are ready. This use of PARTIAL can be seen in Figure 1. Again, it is only possible to create a single binary partition of results this way.

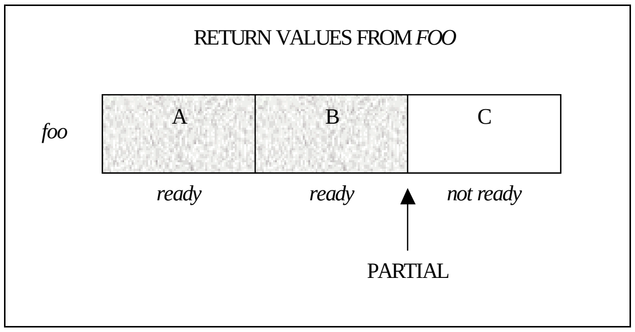

Figure 1. Overloading PARTIAL with additional semantic information

In table entry 2a, we see that very limited progressive extraction of the data is made possible by the addition of the second parameter to EXAMINE. The status of the results can be indirectly derived from the progress of the invocation. This indirect progress indication only applies to the *entire result return set*, which is really only useful if there is only one result parameter or all the result parameters belong tightly together. This is a superset of the web-browser extraction method, actually. With the web-browser extract method for weather data to be computed, images, or simulations , no meta-information about the data is returned to the client (i.e., status *about* the data being 20% complete, etc.). To add such meta-data to web browsing, the browser must be further extended to actively process return values through another mechanism like Java or JavaScript.

In table entry 2b, we see that very limited progressive extraction of the data is again made possible by the addition of the second parameter to EXAMINE. There is really no more power in this examination and extraction model than table entry 2a. However, creative server programmers could take advantage of a scheme similar to that shown in Figure 1. Still, even under such circumstance, programmers are again limited to one predefined subset for extraction and are not allowed the flexibility seen in entries 3b and 4b.

In table entry 3a, we see the addition of per return value examination information. If there is only one return value from a method, the functions of entries 3a, 3b, 4a, and 4b are identical to entries 1a, 1b, 2a, and 2b, respectively. We refer to entry 3a as *semantic partial extraction* because the per result examination allows the user to know exactly when individual return results are ready, but the entire data set must be extracted to get any particular element. This can cause unnecessary delay and overhead, especially with large return data sets.

In table entry 3b, we see the first fully functional partial extraction. Whereas in entries 1b and 2b individual return values could be selected from the return set, partial extraction was not meaningful without a per return value EXAMINE primitive as offered here.

In table entry 4a, progressive extraction is possible, with progress indicators for each data return value. On the other hand, it still suffers from the same overhead as table entry 3a: all data must be extracted at each stage. This expense cannot be avoided.

In table entry 4b, we see the manifestation of the complete CLAM examine/extraction model, one that has all facets of the general result extraction model. Both partial and progressive extractions are possible, including *partial-progressive* extractions (where arbitrary individual elements may be extracted at various stages of completion). Without exception, this model is strictly more powerful than any of the others.

2.4 Granularity in partial extraction

In our discussion of Table 1 we had the special case where it is only possible to extract exactly one subset of parameters (case 1b) and the more general case wherein the partial extraction of an arbitrary set of parameters can be extracted (case 3b). In both cases, the granularity of extraction is given by the individual method parameter. One method parameter is either extracted as a whole, or not at all. If a module supplier wants to provide a very fine granularity for partial extraction, the results of a method have to be split up into as many different parameters as possible. This allows fine granularity for partial examination and extraction, yet has two distinct disadvantages:

- if the result parameters are correlated and together form a larger logical structure, this higher level structure gets lost in the specification of the method as well as in the extraction

- the client is burdened with reconstructing any higher level data structure. This is an overhead concerning programming as well as concerning information dissemination (the client needs to get the information from somewhere how to reconstruct a higher level structure, information that is not necessarily part of the method specification).

add as a figure part of the repository definition in XML with the definition for the parameter PersDat -- leave out the details about short cuts for simple parameters -- from schema spec.

(DOROTHEA'S TREE)

Figure 2. Sample CLAM repository information

There is another way to provide a fine granularity for partial extraction of result parameters without burdening the client with the reconstruction of higher level data structures. The module provider makes the substructure of the result parameter public, and allows users to examine and request only part of a result parameter. One possible way to do that is by using XML for parameters [15]. The structure of parameters are defined by DTDs (or XML-Schemas) and made public along with the method interfaces [16]. For CLAM the DTD for the XML-parameters is defined in the CHAIMS repository as shown in Figure 2. If the client wants to extract parameters as a whole, the client uses the CLAM syntax as discussed in section 2.2. If the client wants to examine and extract just part of a parameter, the client adds an XQL query string to each parameter name in the examine or extract command. For the parameter "PersDat" specified in Figure 2 a client could just examine and extract the element "Lastname" with the following CLAM commands:

```
(lastname_status     =     status)     =
ivh.EXAMINE("PersDat" ; "Lastname")

(lastname  =  "PersDat"  ;  "Lastname")  =
ivh.EXTRACT()
```

XQL is a very simple query language that allows clietns to search for and extract specific sub-elements of an XML-parameter. In the above example, the whole data structure "PersDat" is searched for an element with the tag "Lastname," which is then returned inclusive all of its sub-elements. XQL would also allow clients to specify the whole path (e.g. "/Persdat/Name/Lastname"), or to search for an element anywhere within another element (e.g., "/Persdat//Lastname") or anywhere within the entire parameter (e.g., "//Lastname"). In our specific example, all of these queries return the same data. XQL also allows more complex queries including conditions and subscripts (for more details see [17]).

Using XQL queries for extracting partial results of computational methods should not be confused with using XQL queries to extract data from an XML database, in spite of the apparent similarities. There are several differences:

- here we query non-persistent data; the lifetime of the result parameters is linked to the duration of the invocation

- there exists no overall schema for all the result parameters of one module or even of several modules. The scope of the XML specification (the DTD in the repository) is one single parameter. The relationships between the parameters is not an issue for partial extraction.

- Due to the first two differences, there is also no need or use for a join operation, and a simple query language like XQL fulfills all the needs for partial extraction (whereas for XML-databases more complex query languages like XML-QL might be better suited [18]).

2.5 Implementation issues

2.5.1 Wrapping

Within CHAIMS, all server modules have certain compatibility requirements. Many server modules are actually wrapped legacy code that do not have the necessary components to act as remote servers. For minimal CHAIMS compliance, any legacy module can trivially support an EXAMINE/EXTRACT relationship like that in table entry 1a. This is a single EXTRACT with a per invocation EXAMINE. Simply treat the legacy module like a black box that returns only {DONE, NOT_DONE, ERROR} (without PARTIAL). Also, because return values are collected in the CHAIMS wrapper, the client can freely choose when to request the data, though it must request the data explicitly. The client may also perform multiple requests for the same data without further augmentation of the original code (table entry 1b).

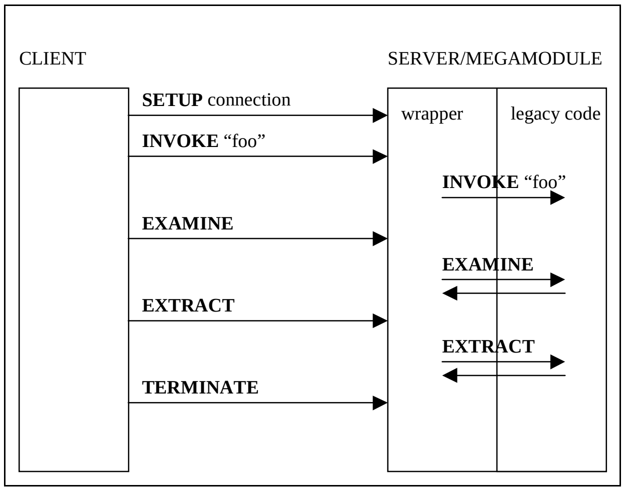

Figure 3. Client-wrapper communication

To use the more powerful models of data extraction, significant modification would be required of naïve modules. We originally classified the two augmentation types (to legacy modules) as either partial or progressive-type augmentations. Partial extraction augmentations are those that make a particular subset of the return data externally available before the completion of the entire invocation. Progressive extraction augmentations are those that post information in multiple stages, i.e., at implementer-defined checkpoints.

Native modules designed for partial and progressive extraction must have a way to post interim results that may be extracted by clients. The interim results must be held in some structure so that request for data may be serviced without interrupting the working code.

The CHAIMS wrapper is a threaded component that handles messages and provides a means of storing interim results and delivering those results to clients. To implement partial and progressive extractions, two pieces of information are required: status and data. When a module posts an interim result (to be delivered by the wrapper or a native server), both pieces of information must be given about the result value.

This status does not need to be provided per method or pre-invocation, however. Such information is extracted from collected knowledge about all partial results. When no partial results are ready, status is NOT_DONE. When all partial results are ready, status is DONE. When some results are ready at any level, per invocation status is simply PARTIAL. When any partial result indicates ERROR, however, the per invocation status should be set to ERROR. This prevents a blind per invocation extraction of all data elements when some may be corrupt.

2.5.2 Result marshalling methods

There are two equally appropriate methods for marshalling partial results, depending upon application: passive and active. The marshalling concerns are basically the servers', not the clients'. With a passive approach, whenever a client specifically requests status or partial results, the message handler requests that information from the working code. The appropriate routines for doing so are provided in the runtime layer (CPAM – Composition Protocol for Autonomous Megamodules) in the form of a Java class, or they may be user developed. Of course, native codes written with the intent of posting partial results are easier to work with than wrapped codes and can use any suitable language (Java, or otherwise). Figure 4 better shows the marshalling and examination interaction between a wrapper and legacy code. The status and progress information is held in the wrapper. Also, temporary storage locations for the extractable parameters are located in the wrapper.

The active approach to data marshalling is more appropriate for certain problem types. In this method, when a server program reaches a point where it is appropriate to post status and results, it does so, directly to the CPAM objects or wrapper layer. The trade-offs between the approaches should be clear. Active posting is conceptually simpler and easier to code, but requires the overhead posting and storing interim results that may never be examined. Figure 4 shows how an active approach to data marshalling would proceed through time. After the wrapper receives an EXAMINE request, the appropriate routines actively inspect the legacy code to update the status/progress structure in the wrapper. After the EXTRACT is received, the request is passed to the legacy code, and the data structures are then updated, before results are passed back to the client.

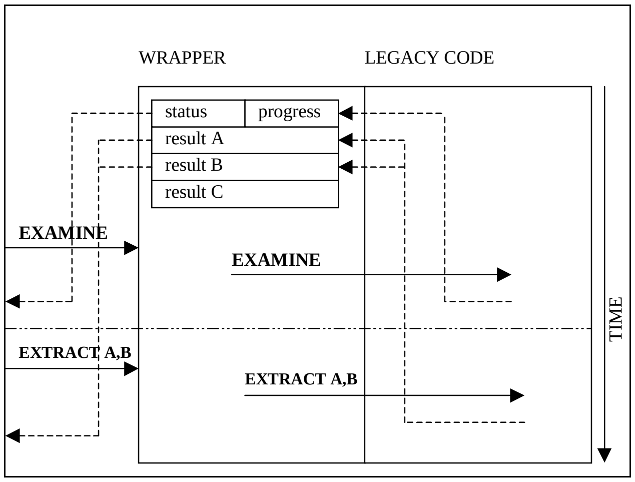

Figure 4. Wrapper result marshalling

2.5.3 Termination

The ability to delay extractions and make repeated extractions implies that a server no longer knows exactly when a client is finished with an invocation. With a traditional RPC, this was not a problem. In that case, when the work was complete, results were returned, and the server (or procedure) had no further obligations to the client. With arbitrary extraction, the server is obligated to hold data for the client.

Even without allowing repeated extractions, there are more subtle reasons for which the server must hold data for clients. In the case of a partial extraction from our example method *foo*, a client may extract result fields A and B, but the server does not *know* that the client is not also interested in result field C. Since there is no a priori communication of intent from client to server, their relationship must be changed somewhat.

The obligation for a server to cache and store results is balanced by a client's obligation to explicitly terminate an invocation. This explicit termination merely signals to a server that a client is no longer interested in further extractions from a particular invocation, but is an integral detail of this model of result extraction.

2.5.4 Repository

There should be a repository of method interfaces and the structure of return values available to programmers using this arbitrary extraction model. Of course, when programming "in the small," (i.e., stand-alone programs, in-house projects, etc.), this is not really an issue at all. When making services available for sale/use externally, service providers must provide the appropriate information about the results which can be extracted. For instance, if delivering *foo* over the net, a provider should indicate to users that fields A, B, and C may be extracted separately. This information in the context of CHAIMS (where we assume "programming in the large") is provided via a repository.

3. COMPARISONS

In the following we compare the extraction model as defined in CLAM and mirrored primitive by primitive in the CHAIMS access protocol CPAM to the extraction models found in the following protocols:

- web browsing
- JointFlow
- SWAP
- CORBA-DII

3.1 Partial and progressive result extraction in web browsing

Web browsing generally falls into the category of services that we refer to as data services. Recall, data services primarily deliver specific data requested by clients, rather than computational services which add value to client supplied data. Clients are usually represented by one of many available web browsers or crawlers while web servers deliver data to those clients. Clients request data using the http protocol. Data extracted (documents delivered) from servers are often written in html, and often have components of varying types, including images, audio, and video.

Web browsing occurs in batch and interactive ways. Batch browsing is performed by crawlers for many reasons, such as indexing, archiving, etc. Interactive browsing is performed by humans for numerous reasons, such as information gathering, electronic commerce, etc. A browser of either sort makes a request to a server for a specific document. That document is returned to the client, and serves as a template for further requests. If the document is html, the browser may parse the document and determine that there are other elements that form the complete document (i.e., images tags). The document serves as a schema, describing the other elements that may be extracted from the server.

After a main document has been fetched, we can consider the possible partial and progressive extractions that can take place. To extract a sub-element of a web page, an http request is sent to a server, and the data or an error message is returned. In batch browsing, the textual information contained in the page is frequently enough to be meaningful. This is very different from the generalized result extraction model we discuss in that the schema of the results is not meaningful in itself. But, in web-browsing, the page retrieved is often meaningful, not just for the sub-elements it describes. This aside, we consider result extraction in terms of gathering sub-elements from pages

In interactive browsing, partial extraction is a simple process, and is at least marginally exploitable in the most widely used interactive browsers (Netscape and Microsoft's Internet Explorer). Both feature an "auto-load" feature that can be toggled (to varying degrees) to automatically load (or *not* load) different content type such as images, audio, or video. For instance, some users are concerned with images and text, but do not with to be disturbed by audio. Their browser makes the http requests for all sub-elements, save audio. This is partial extraction. In other cases, especially with slower internet connections, images are expensive to download, users may choose to not automatically download images until determining that a particular image or set of images is important enough to invest time in.

Partial extraction in web-browsing is a special case of the general partial extraction model in that the first result to be extracted always contains information about the other results to be extracted. Based on this first result, the client not only determines its interest in the other elements of the page, but also gets the information about what other results are available at all. This is in contrast to our general model, where a result parameter may but need not provide information about other result parameters, and where all possible result parameters are specified in a repository beforehand.

The most commonly found progressive extraction in web browsing is quite different from progressive extraction in a computational service though progressive extraction of computational services over the web, e.g. improving simulation data, is also feasible. In a computational service, progressive extraction refers to extracting various transformations of input data over the life of a computation. In web browsing, progressive extraction is actually repeated extraction of a changing data stream. Weather services on the web often provide continuous updates and satellite images. Stock tickers provide updated information so users can have current information about their investments. Repeated extractions from the same stream show the stream's progress through time. Sometimes these repeated extractions may be done by manually reloading the source, or they may be pulled from servers by html update commands, JavaScript's, embedded Java-code, etc. Such data is retrievable any time and its progress status is always DONE and 100% accurate, yet we expect the data to contain also information about to which point of time it refers.

3.2 Incremental result extraction and progress monitoring in JointFlow

JointFlow is the Joint Workflow Management Facility of CORBA [6]. It is an implementation of the I4 protocol of the workflow reference model of WfMC [11] on top of CORBA. JointFlow adopts an object oriented view of workflow management: processes, activities, requesters, resources, process managers, event audits etc. are distributed objects, collaborating to get the overall job done. Each of these objects can be accessed over an ORB, the JointFlow specification defines their interfaces in IDL. We have chosen JointFlow for comparison as it is a protocol that also support the request of remote computational units which can yet need not to have some degree of autonomy, and the protocol is also based on asynchronous invocation of work and extraction of results, having special primitives for invocation, monitoring, and extraction.

Work is started in that a requester asks a process manager to create a new process. The requester then communicates directly with the new process, setting context attributes in the process and invoking the start operation of the process. A process my be a physical device, a wrapper of legacy code, or it may initiate several activity objects which might in turn use resources (e.g. humans) via assignments or itself act as requesters for other processes. Our focus of interest here is the interaction between the requester and the process concerning result extraction and progress monitoring.

3.2.1 Monitoring the Progress of Work

Both, processes and activities are in one of the following states: running, not_running.not_started, not_running.suspended, completed (successfully), terminated (unsuccessfully), aborted (unsuccessfully). A requester can query the state of a process, the states of the activities of the process (by querying and navigating the links from processes to activities), and the states of assignments (by querying and navigating the links from activities to assignments). If the requester knows the workflow model with all its different steps implemented by the process, the requester might be able to interpret the state information of all subactivities and assignments and figure out what the progress of the process is. If the model is not known, e.g., due to an autonomy boundary as they are assumed in CHAIMS, the only status information provided by the JointFlow protocol itself is essentially completed or not yet completed. In contrast, CHAIMS supports the notion that certain services may support progress information (e.g. 40% done) that can be monitored. This information is more detailed than just running or complete, and more aggregated and better suited for autonomous services than detailed information about component activities.

In contrast to CHAIMS that polls all progress information, in JointFlow a process signals its completion to the requester by an audit event. These **audit events** could also be used to implement CHAIMS-like progress monitoring on top of JointFlow: a process can have a special result attribute for progress information and the process is free to update that attribute regularly. It then can send an audit event with the old and new value of the progress indicator result to its requester after each update. Yet this result attribute cannot be polled by a requester (in contrast to CPAM and SWAP), because get_result only returns results if all results are available at least as intermediate results.

3.2.2 Extracting Results Incrementally

Both, processes and activities have an operation *get_result():ProcessData* (returning a list of name value pairs). Get_result does not take any input parameter and thus returns all the results. The get_result operation may be used to request **intermediate result data**, which may or may not be provided depending upon the work being performed. If the results cannot yet be obtained, the operation get_result raises an exception and returns garbage. The results are not final until the whole unit of work is completed, resulting in a state change to the state complete and a notification of the container process or the requester. This kind of extracting intermediate results corresponds to the progressive extraction of all result attributes in CHAIMS. The following features found in CHAIMS are **not available** in JointFlow:

- **Partial extraction with get_result**: only all or none of the result values can be extracted by get_result, and there is no mechanism to return an exception only for some of the values.
- **Progressive extraction with get_result of just one result attribute** when not yet all other results are ready for intermediate or final extraction
- There is no **accuracy information** for intermediate results, unless it is in a separate result attribute. There is no possibility to find out about the availability or the accuracy of intermediate results unless requesting these results.

Though partial and progressive result extraction are not part of the design of JointFlow, they also can be achieved by using audit events and by pushing progressive and partial results onto the requester, instead of letting the requester poll for them. A process or an activity can send out an audit event to its requester or to the containing process whenever one of the result values has been updated. This event would then contain the old as well as the new result value. In case of large data and frequent updates, this messaging mechanism could result in huge amounts of traffic. The mechanism would have to be extended by special context attributes that tell the process or activity in advance which results should be reported in which intervals. Yet this results in a very static and server centric approach, in contrast to the client-centric approach in CHAIMS that is based on data on demand. Also, as partial and progressive result extraction are not mandated by the JointFlow protocol itself, it is questionable how many processes and activities would actually offer it.

3.3 Incremental result extraction and progress monitoring in SWAP

SWAP (Simple Workflow Access Protocol) is a proposal for a workflow protocol based on extending http. It mainly implements I4 (to some extend also I2 and I3) of the WfMC reference model. SWAP defines several interfaces for the different components (which are internet resources) of the workflow system that interact via SWAP. The three main components are of type ProcessInstance, ProcessDefinition and Observer. The messages exchanged between these components are extended http-messages with headers defined by SWAP. The data to be exchanged is encoded as text/xml in the body of the message.

A process instance (having the interface ProcessInstance) is created and started by sending a CREATEPROCESSINSTANCE message to the appropriate ProcessDefinition resource. This message also contains the context data to be set and the URI of an observer resource that should be notified about completion and other events. The response contains the URI of the newly created process instance. The process is started either automatically by the ProcessDefinition resource if the CREATEINSTANCEMESSAGE contains the startImmediately flag, or by sending a PROPPATCH message to the process instance with the new state running. A process instance resource can delegate work to other resources by acting itself as an observer and ask some ProcessDefinition resources for the creation of other process instances. As in JointFlow and in CHAIMS, the process instance creation, setting of context attributes, start of the process, and the extraction of results are done asynchronously.

3.3.1 Result Extraction and Result Monitoring

Results are extracted from a process instance by sending it the message PROPFIND at any time during the execution of a process instance. This message either returns all available results, or if it contained a list of requested result attributes, it only returns the selected ones. Only result attributes are returned that are available. If requested attributes are not yet available, presumably an exception should be returned for these result attributes. SWAP does not specify if the results returned by PROPFIND have to be final or not. Given the possibility to ask for specific result attributes, and to get exceptions for specific result attributes in case they are not available (made possible by having exceptions encoded in XML instead of having just one possible exceptions for one procedure call as in the CORBA based JointFlow protocol), allows some degree of partial and maybe even progressive extraction.

A process instance signals the completion of work to an observer with the COMPLETE message. This message also contains the result data: all the name value pairs that represent the final set of data as of the time of completion. After sending the COMPLETE message, the resource does not have to exist any longer, this in contrast to CHAIMS where the no result data is lost until the client (observer) sends a TERMINATE.

A process instance can also send NOTIFY messages to an observer resource. These messages transmit state change events, data change events, and role change events, data change events containing the names and values of data items that have changed.

3.3.2 Incremental Result Extraction as Defined in the CHAIMS Model

The mechanisms of SWAP allow the following kind of result extraction and progress monitoring:

- **Partial result extraction**: Either pushing results via NOTIFY messages or pulling results via PROPFIND messages is possible. NOTIFY sends all new result data, PROPFIND returns all available result data whether or not they have already been returned by a previous PROPFIND. Notification of result changes without sending also the new values is not possible unless additional result attributes are added. The same is true for getting the status of individual results: asking for the status of results without getting also the results, is not possible unless a state attribute is added for each data attribute to the set of result attributes.
- **Progressive result extraction**: The SWAP specification does not explicitly specify if progressive result updates in a process instance are allowed or not. If not, the result attributes would not be available until their values are final. If yes, then progressive results can be extracted either by pushing results via NOTIFY messages or by pulling results via PROPFIND messages. Accuracy indication is not provided, it would have to be implemented via additional result attributes.

3.3.3 Process Progress Monitoring

PROPFIND not only returns all result values available, it also returns the state of the process instance and additional descriptive information about the process. As possible states can be specified by the process itself, PROPFIND also returns the list of all possible state values, yet in most cases it would probably just be not_yet_running, running, suspended, completed, terminated, etc (the basic set of states defined by I4). A process instance can be asked for the URI of all the processes it has delegated work to, and an observer then can directly ask this subprocesses about their statuses. This is analogue to the model found in JointFlow, and thus has the same drawbacks concerning autonomy and concerning amalgated progress information.
Overall progress information is not specified by SWAP, but it could be implemented by a special result attribute assuming that result attributes can be changed over time. Such result attributes could be extracted any time by PROPFIND, independent of the availability of other result attributes.

Though SWAP does not support incremental result extraction as defined in CHAIMS, it could quite easily either be added to the SWAP protocol itself or done by using the SWAP protocol as defined and applying the simple workarounds mentioned above. As SWAP has very similar goals in accessing remote processes as CHAIMS, and as it is a very open and flexible protocol, its result extraction model is already very close to the one of CHAIMS and could be easily extended to contain all aspects of the CHAIMS extraction model. Yet as SWAP has not been designed with incremental extraction in mind, it does not have the strong duality between extract and monitoring command as found in CHAIMS between EXAMINE and EXTRACT.

3.4 Incremental result extraction and progress monitoring in CORBA

3.4.1 CORBA- DII

CORBA offers two modes for interaction between a client and remote servers: the static and the dynamic interface to an ORB. For the static interface an IDL must exist that is compiled into stub code that can be linked with the client. The client then executes remote procedure calls as if the remote methods were local.

The dynamic invocation interface (DII) offers dynamic access where no stub code is necessary. The client has to know (or can ask for) the IDL from the remote object, i.e., the names of the methods and the parameters they take. The client then creates a request for a method of that object. In this request the method name appears as a string and the parameters appear as a list of named values, with each named value containing the name of the parameter, the value as type any (or a pointer to the value and a CORBA type code), the length of the parameter, and some flags. Once the request is created, the method can be invoked. This is either done synchronously with invoke or asynchronously with send (in fact, some flags allow more elaborate settings). Invoke returns after the remote computation has completed, and the client can read all OUT parameters in the named value list. In case of a send, the client is not blocked. In order to figure out when the invocation has finished, the client can use get_response, either in a blocking (it waits until invocation is done) or a non-blocking mode. As soon as the return status of get_response indicates that the remote computation is done, the client can read OUT parameters from the named value list.

In case of the asynchronous method invocation in CORBA-DII, the progress of an invocation can be monitored and asked for by the client as far as completion is concerned, but no further progress information is available. Progressive extraction of results is not supported by DII. Of course a client is free not to read and use all results after the completion of an invocation, yet while the computation is going on no partial extraction is supported.

3.4.2 CORBA Notification Service

In order to mimic the incremental result extraction of CHAIMS, one could use asynchronous method invocation with DII coupled with the event service of CORBA. The client could be implemented as a PullConsumer for a special event channel CHAIMSresults, the servers could push results into that channel as soon as they are available, together with accuracy information. Though event channels could be used for that purpose (we could require that every megamodule uses event channels for this), an integration of incremental result extraction and invocation progress monitoring into the access protocol itself is definitely more adequate when we consider this to be an integral part of the protocol. The same is true for the languages used to program the client: while CLAM directly supports incremental extraction and progress monitoring, this is not the case for any of the languages in used for programming CORBA clients.

4. CONCLUSIONS

In the CHAIMS project, we sought to build a composition-only language for remote, autonomous services. To do this, we had to consider many different extraction models used in different domains. This examination led to the realization that a simple asynchronous RPC-style approach was not enough.

To build a language and an access protocol to support arbitrary result extractions took careful consideration of the myriad ways extractions were currently being used in widespread as well as hand-crafted systems. Building support and primitives for all of these result extraction methods (autonomously, progressively, and partially) and then binding them within one system has led to the formulation of a comprehensive model for arbitrary result extraction.

Our model captures the notions of traditional result extraction, partial extraction and progressive extraction. By combining two simple primitives in CLAM (EXAMINE and EXTRACT), the full power of each of these extraction types can be achieved. This extraction model is appropriate as a template for existing systems, and future languages as well. It is generic to result extraction, and only assumes that the necessary asynchrony can be achieved among components through distributed communication, threading, or other available means.

5. REFERENCES

[1] D. Beringer, C. Tornabene, P. Jain, and G. Wiederhold: “A Language and System for Composing Autonomous, Heterogeneous and Distributed Megamodules,” DEXA 98: Large-Scale Software Composition, Vienna, August 1998.

[2] B. Chapman, M. Haines, P. Mehrotra, H. Zima and J. Van Rosendale, “Opus: A Coordination Language for Multidisciplinary Applications,” ICASE Technical Report 97-30, June 1997.

[3] M. Haines and P. Mehrotra, “Exploiting Parallelism in Multidisciplinary Applications Using Opus,” *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing*, San Francisco, CA, February 1995.

[4] J. Hanly, E. Koffman and J. Horvath, *Porgram Design for Engineers*, Addison-Wesley, Menlo Park, CA, 1995.

[5] ICASE Research Quarterly, Vol. 4, No. 1, March 1995.

[6] Workflow Management Facility, Revised Submission, OMG Document Number: bom/98-06-07, July 1998.

[7] L. Melloul, D. Beringer, N. Sample and G. Wiederhold, “CPAM, A Protocol for Software Composition,” CAiSE'99, Heidelberg, Germany, June 1999 (Springer LNCS).

[8] N. Sample, D. Beringer, L. Melloul and G. Wiederhold: “CLAM: Composition Language for Autonomous Megamodules,” Third Int'l Conference on Coordination Models and Languages, COORD'99, Amsterdam, April 26-28, 1999 (Springer LNCS).

[9] Simple Workflow Access Protocol (SWAP), Keith Swenson, IETF internet draft, August 1998.

[10] G. Wiederhold, P. Wegner, and S. Ceri: “Towards Megaprogramming: A Paradigm for Component-Based Programming”; Communications of the ACM, 1992(11): p.89-99.

[11] Workflow Management Coalition: The Workflow Reference Model, Document Number TC00-1003, Nov 1994.

[12] C. Bartlett, M. Haines, and N. Sample: “Pipeline Expansion in Coordinated Applications,” International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA’99), June 1999.

[13] T. Pratt and M. Zelkowitz, Programming Languages, Design and Implementation, 1996, Prentice Hall, Inc.

[14] G. Wiederhold, D. Beringer, N. Sample and L. Melloul, “Composition of Multi-site Services,” 4th World Conference on Integrated Design and Process Technology IDPT'99, Kusadasi, Turkey, June 1999.

[15] T. Bray, J. Paoli and C. Sperberg-McQueen, “Extensible Markup Language (XML) 1.0,” W3C Recommendation, February 1998.

[16] A. Davidson, et al., “Schema for Object-Oriented XML 2.0,” W3C Note, July 1999.

[17] J. Robie, “XQL Tutorial,” March 1999 (http://metalab.unc.edu/xql/xql-tutorial.html).

[18] [18] A. Deutsch, et al., “XML-QL: A Query Language for XML,” submission to the W3C, August 1998.