ELSEVIER

Computer Networks 34 (2000) 467–480

COMPUTER NETWORKS

www.elsevier.com/locate/comnet

A new multi-search engine for querying data through an Internet search service on CORBA

Yue-Shan Chang [a,*], Shyan-Ming Yuan [a], Winston Lo [b]

[a] *Department of Computer and Information Science, National Chiao Tung University, 1001 TA Hsueh Road, Hsin-Chu 30050, Taiwan, ROC*

[b] *Department of Computer and Information Science, Tung Hai University, Taichung, Taiwan, ROC*

Received 14 February 2000; received in revised form 5 May 2000; accepted 19 May 2000

Abstract

Search engines are important but generally far from ideal tools of the World Wide Web (WWW). Many researchers therefore prefer to use meta-brokers to construct multi-search engines (MSE). However, these have no uniform programming interfaces, which makes tying them with other search engines difficult. Moreover, for an application that needs a search service capability, querying them is difficult.

To reduce that difficulty, we propose in this paper an Internet search service (ISS) based on common object request broker architecture (CORBA) that follows the style of common object service specification (COSS). We design a multi-search engine based on ISS, which we term Octopus. For a system developer, because of its uniformity of interface, Octopus easily ties with any search engine. Equally, for an application programmer, the ISS offers a clear interface for application programs to search for information or mine data from the Internet. We demonstrate our approach to designing multi-search engines through ISS by tying two search engine agents, Yahoo and AltaVista, with Octopus and show how CORBA clients query them. Programmers may use this interface to construct their search engine agents or query a search engine agent in their applications.

Keywords: Internet; Multi-search service; Search engine; World Wide Web; CORBA

1. Introduction

Recent advances in the computer network and the Internet, have given the Web increasing popularity, but the growth in Web sites has made searching the Internet a more involved task, with the result that search engines now claim more importance as tools, and their number increases accordingly.

Standard search engines, for example, Yahoo, [1] AltaVista, [2] Lycos, [3] InfoSeek, [4] Galaxy, [5] and WebCrawler, [6] may help users find what they

* Corresponding author. Tel.: +886-3-557-2930; fax: +886-3-559-1402.

E-mail address: ysc@mhit.edu.tw (Y.-S. Chang).

[1] http://www.yahoo.com.
[2] http://www.altavista.digital.com.
[3] http://www.lycos.com.
[4] http://www.infoseek.com.
[5] http://galaxy.einet.net/galaxy.html.
[6] http://webcrawler.com.

1389-1286/00/$ - see front matter
PII: S1389-1286(00)00131-6

want, but they have their limitations. For example, none of them individually is sufficient and most of them return too many irrelevant, outdated, or unavailable references [1]. In general, users have to query many engines before obtaining the most relevant matches. In addition, each search engine has its own interface. A novice is confused by this variety of search engines. Designing from scratch a comprehensive search engine with a search capability equal to other search engines combined is not easy. Many researchers thus construct MSEs that use meta-brokers, such as MetaCrawler [1], Customizable Multi-Engine Search Tool [2], SavvySearch [3,4], Softbot [5], Amathaea [6], and the solution proposed by Overmeer [7,8]. These tools tie a number of search engines together in a system and act as a dispatcher. When a user initiates a search, a MSE dispatches the request to various search engines and collects the results.

1.1. Motivations and objectives

Even with MSEs, however, there are problems, especially with the scalability and flexibility of a system. Even if an MSE is capable of tying with any existing search engine, it may be necessary in the future to integrate it with new and more powerful search engines. Most MSEs lack extensibility, because they have no uniform interface for standard search engine agents. Generally, there are problems with tying an MSE to a new search engine.

Although Internet users may find MSEs useful, this is less true for programmers, who need to have a search component included in their applications. When an application needs to conduct a search, the programmer must either design a search component in their applications or query existing search engines. Regardless of which approach is used, the programmer will always need to extract what is wanted from the complicated HTML file that is returned. For example, a programmer developing an application that has a search capability must explore and analyze the interface of the search engine. Then, the query string must be encapsulated into URL format in terms of the query interface of the search engine and be sent to the search engine in HTTP protocol. When the results are returned, the program must also extract the information from the complicated HTML file. Each operation needs to handle the network connection. These are tedious tasks. An MSE is thus difficult to use when a specific application must search.

Among programming techniques, object orientation [9] offers greater portability and reusability and has been widely applied to software design and development. Many new applications are designed in object-oriented language, such as C++ or Java. In recent years, a range of distributed object middle-ware has become available, for example, the Object Management Group's CORBA [10], Microsoft's DCOM [11] and Sun's JAVA RMI. CORBA is an industrial standard and has more than 1000 members worldwide. Moreover, it has been ported to many operating environments, such as Microsoft's Windows, UNIX and MVS.

Here, we propose an ISS on CORBA, which is an industrial standard of a distributed object-oriented platform [10], and design a multi-search engine based on ISS. The design of the ISS interface follows the style of COSS [12].

Several reasons motivated this work. First, every search service on the Internet has its own interface, which is confusing to novices. We therefore propose a uniform interface that can accommodate most of the interfaces found in search engines. Second, if an application requires to conduct a search, then incorporating search services must be made easy. By means of ISS, programmers can use this interface to construct search components or to query the search engines in their applications. Based on the CORBA standard, applications can be developed in other environments, such as CORBA, COM, and Java, as shown in Fig 1, but those so developed must be mediated through IIOP bridges [10,13]. Third, since the interface is uniform, designing a multi-search engine based on ISS is simple. Fourth, we hope to establish a standard for Internet Search Service on the OMG's COSS. Finally, we believe that ISS can be easily applied to other types of search engines, such as knowledge-discovery systems, real-estate systems, and digital libraries, and to heterogeneous search engine agents. Thus, for most search or query services, whether or not on

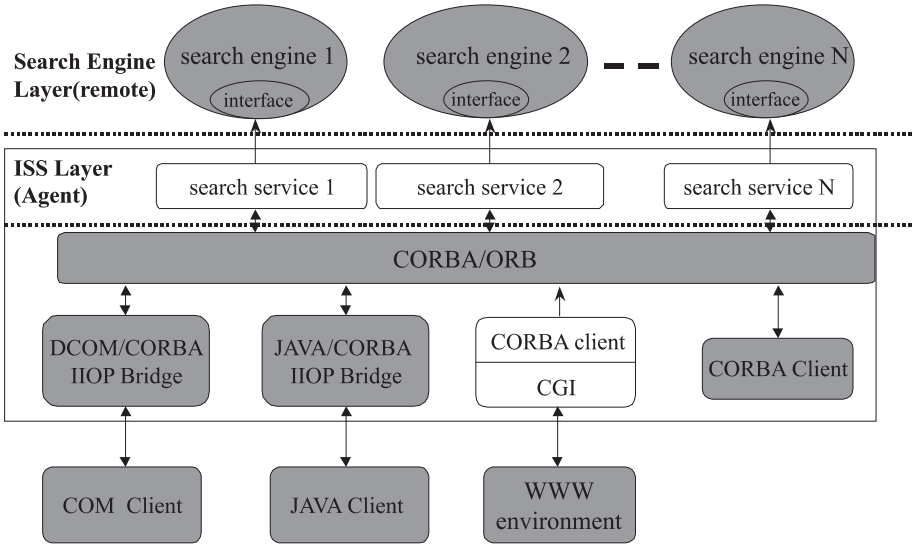

Fig. 1. Internet search service architecture.

the Internet, this approach can be extended to multi-search engines, the design for which can inherit the advantages described above.

The major objective of this paper is to propose a uniform interface for Internet search services, which will offer a programming interface for locating of what it wishes to retrieve. Since the architecture of ISS is of a 3-tier client/server model and its interface is uniform, any specific application that needs to search for information or to mine data from the Internet need only initiate the search operation of an agent via ISS. Programmers do not need either to explore the interfaces of various search engines, or to construct search components in their applications. In Section 2, an ISS interface and a programming example of ISS are described. In Section 3, we build our experimental multi-search engine, which we term Octopus, and demonstrate the proposed approach to designing a multi-search engine using ISS.

We begin by exploring the interfaces and attributes of existing search engines and then combine these interfaces into a uniform interface to form an ISS. Then, using the ISS interface, we construct two representative search engine agents – one for Yahoo and the other for AltaVista. As the ISS is based on CORBA, and the agents are built as Internet search components, programmers can use the interface to search Web sites in their applications. In addition, we tie the agents together and build Octopus. It is built as a multi-threaded and a multi-agent version to serve incoming requests, then to collate and filter results before returning them to users. Although the agents were constructed as CORBA objects, and the interface was uniform, other search engines could be easily tied to Octopus.

1.2. Search engine overview

Search engines are powerful tools for assisting users to navigate the rapidly expending World Wide Web. Most large-scale search engines can be divided into two categories: directory scheme, such as Yahoo and Yam,[7] and active search scheme, such as AltaVista and InfoSeek, etc. With directory scheme, the Web manager must register the Web's address, description, and other identifying information, while active search schemes search Internet Web sites periodically and index related items of information in the database.

In directory schemes, registered Web sites are categorized manually. Querying this kind of search

[7] http://taiwan.iis.sinica.edu.tw/b5/yam.

engine may produce more relevant results. Since directory schemes can only accommodate registered sites, and many sites are unregistered, results from them may be fewer than those obtained from active search schemes. In comparison, active search schemes search most Web sites periodically and so return more results. Nevertheless, they do not guarantee relevance, even if the results have a higher index. However, combining the advantages of both search schemes could improve search performance and results.

Most search engines have their own interfaces, which may not be alike. We investigate those interfaces to define our ISS interface. Two representatives search engines are explored – Yahoo and AltaVista – and the results are shown in Table 1. A full description of the interfaces of both engines can be found on their home page.

There are two types of interfaces: common and specialized. Common interfaces contain many attributes that are included in most search engines; specialized interfaces do not. These two types are shown in Table 1, and we believe that they include most existing search engine interfaces. Nevertheless, we have not covered all search engines so far, but we shall add to these interfaces, and try to include as many engines as possible. The design of our ISS is described in Section 2.

The paper is organized as follows. Section 2 presents the design of our ISS based on CORBA and a sample program to demonstrate how to use it. Section 3 describes an experimental multi-search engine based on our architecture, which we term Octopus. Section 4 presents a discussion. Finally, Section 5 gives conclusions.

Table 1
Interface comparison for Yahoo and AltaVista

	Yahoo	AltaVista
Common interfaces		
Include	+	+
Exclude	–	–
Wildcard	*	*
OR	Space	Space
Exact Phrase	“ ”	“ ”
Title	t:	Title:
URL	u:	url:
Specialized interfaces		
Anchor		Anchor:
Applet		Applet:
Domain		Domain:
Host		Host:
Image		Image:
Link		Link:
Text		Text:
Near		~
Date restriction	1 day ago	dd/mmm/yy
	3 days ago	
	1 week ago	
	1 month ago	
	3 months ago	
	6 months ago	
	3 years ago	
Display matches	10	
Per page	20	
	50	
	100	
Search area	Yahoo	
	Categories	
	Web site	
Search database	Yahoo!	Usenet
	Usenet	

2. Designing an Internet search service

In devising a multi-search engine, we first define an ISS and describe its design in detail before presenting an example to demonstrate its use.

2.1. Architecture of ISS

We define an ISS interface by following the style of COSS. One of major objective of our ISS is to provide a uniform interface for search engines. Programmers can use this interface to construct search engine agents or to query search service in their applications based on our object implementation. We design the ISS based on the description of the interface comparison given in Section 1.2. The design consists of three components: *SearchFactory*, *Search*, and *ResultCollection*. The *Search* is at the core of executing a search in ISS. The *SearchFactory* component creates *Search*. The *ResultCollection* component collects results. Fig. 2 shows the relationship of these components. An arrow with a vertical bar is used to show that the target object supports the interface named next to the arrow and that clients holding an object reference of this type can perform operations defined

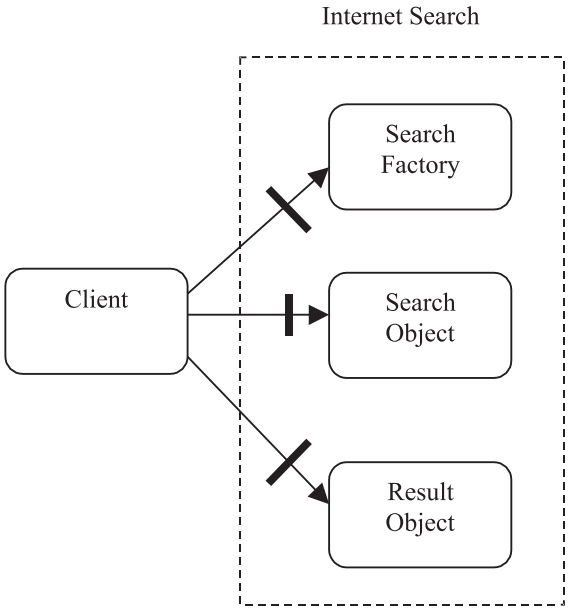

Fig. 2. Structure of interface search service.

by the interface. Our design includes as many search interfaces as possible. Next, we describe the ISS interface and its processing scenario.

2.2. The ISS interface

The *SearchFactory* creates a *Search* object. Before a client program obtains a *Search* object, it must bind to the *SearchFactory* to obtain its object reference. This interface is shown in Table 2.

After the client program obtains the reference of the *SearchFactory* object, it then invokes *NewSearch*( ) operation to obtain the *Search* object, which is the component to invokes which search engine agents. In ISS, the *Search* interface have five methods, which are *AddKeyword*( ), *RemoveKeyword*( ), *GetKeyword*( ), *ExecuteSearch*( ), and *AbortSearch*( ). These operate as follows.

Table 2
The *SearchFactory* interface

```
interface SearchFactory          //search factory interface
  {
    Search NewSearch();
  };
```

When a client program obtains the reference of the *Search* object, it then may perform *AddKeyword*( ) operation to put query string into the *Search* object. The query string is removed by the *RemoveKeyword*( ) operation. These two methods have an input keyword parameter in respect of the added or the removed string. Each keyword consists of a query string and an attribute. The attribute denotes whether the string is included or not.

In addition, a client may look for the query string by the *GetKeyword*( ) operation, with an input parameter that is the needed key word index. Before invoking a search engine agent, a client must put one or more attributes related to the query into the *Search* object. Then, the client program initiates the *ExecuteSearch*( ) operation to perform the search. To discontinue a search, the client program can perform the *AbortSearch*( ) operation. These methods are shown in Table 3.

To accommodate all search engine interfaces, we refer to Table 1 and define a few attributes in the *Search* interface. These attributes are *Domain*, *Tag*, *Date*, *Near*, *DispNum*, *Area*, *DataBase*, and a read-only *No_Keyword*. Most of them are bit-wise representations. For instance, a *Tag* attribute can represent the search target which may be in the form of *title*, *URL*, *anchor*, *applet*, *image*, *link*, or *text*. We declare a long integer to represent this attribute. Each bit represents a tag, and the remainder bits are reserved for future use. In CORBA, attributes can be translated into two operations by an IDL compiler. They are *get_xxx* and *set_xxx*, respectively. Here, xxx is the attribute's name. How to use these attributes is implementation-dependent.

The third ISS interface is the *ResultCollection*, which is shown in Table 4. In this interface, the *retrieve_element_at*( ) is the only method function. When a client program performs the *ExecuteSearch*( ) operation, the *Search* object issues a query request to related search engines, creates and returns the *ResultCollection* object reference to the client program. The client program reads the number of results from the read-only attribute named *No_Result*, and then retrieves the returned references that include a data structure consisting of *Title*, *URL*, *Description*, *Date*, and *Weight*,

Table 3
The Search interface

```
struct Keyword                                  //keyword structure
{
   char Inclu_Exclu;
   string item;
};
interface Search                                //search interface
{
   attribute string Domain;                     //domain and host
   attribute long Tag;                          //title, URL, anchor, applet, image, link, text
   attribute string Date;
   attribute long Near;                         //how many words between two string
   attribute unsigned long DispNum;             //how many references displayed in a page
   attribute char Area;                         // Web site, Categories
   attribute boolean DataBase;
   readonly attribute long No_Keyword;          //get the number of added keyword
   boolean AddKeyword(in Keyword add);          //add keyword for future search
   boolean RemoveKeyword(in Keyword removed);   //remove added keyword
   Keyword GetKeyword(in long index);           //get added keyword
   ResultCollection ExcuteSearch( );            //execute search
   Boolean AbortSearch( );                      //abort this search
};
```

using *retrieve_element_at*( ) method. In the following two subsections, we explain the scenario of ISS and present a programming example.

2.3. Normal scenario of ISS

Fig. 3 shows the normal scenario for ISS. For the client program, it first binds to the *SearchFactory* to get its object reference, and execute the *NewSearch*( ) operation to construct a new *Search* object. When the *SearchFactory* receives a *NewSearch* request, it creates a *Search* object and returns object reference to the client. Once the client has this, it can execute other operations, such as *AddKeyword*, *RemoveKeyword*, and *GetKeyword*, and set related attributes into the *Search* object. Then, the client program uses the *ExecuteSearch* method to issue a search request. Once the *Search* object receives a request, it encapsulates related keywords and attributes into the search engine query string and sends it to corresponding search

Table 4
The *ResultCollection* interface

```
struct Result                   //result structure
{
   string Title;
   string URL;
   string Description;
   string Date;
   float weight;
}
interface ResultCollection      //result collection interface
{
   readonly attribute long
No_Result;
   Result retrieve_ele-
ment_at(in long where);
};
```

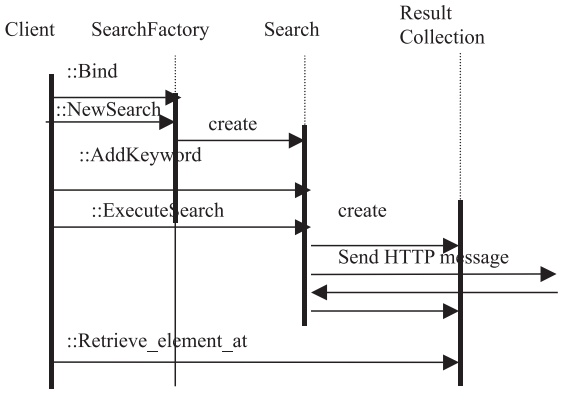

Fig. 3. Scenario of ISS architecture.

engines. When the *Search* object receives the search result from the search engine, it de-encapsulates returned messages and puts them into *ResultCollection* object. Then, the *Search* object returns *ResultCollection* object reference to the client program, which then extracts query results.

2.4. Programming example

In this subsection, first we demonstrate the codes for a client program and show how to query search engine agents. According to the described in Section 2.2 interface, we first compile the interface definition language (IDL) file and generate the client stub and server skeleton. Then, we implement the search engine agent component and compile it with the server skeleton. Finally, we write a server program and register it into ORB. The server program is shown in Table 5.

As described in Section 1, an application program using this interface can easily search for information from the Internet. This functionality is not supported by other MSEs. We now demonstrate the implementation of a client program. The code sequences shown in Table 6 are the same as the scenario for ISS.

First, the client must bind with a search engine agent and obtain a *SearchFactory* object reference. If the system supports multiple agents, then the client may also bind with other agents and set the attributes of agent. Second, the *NewSearch*( ) operation is invoked to obtain the *Search* object reference. Third, the client puts the query string into the Search object by invoking the *AddKeyword*( ) operation. Fourth, all the attributes are set. Fifth, the search is executed by invoking the *ExecuteSearch*( ) operation. Finally, the result is obtained from the *ResultCollection* object.

Table 5
The server program

```
int main(void)
{
InternetSearchService_SearchFactory_impl ISS_YAHOO;
try {
   CORBA_Orbix.impl_is_ready("ISS_YAHOO");
} catch (CORBA_SystemException &SysEx) {
   ...
}
return 0;
}
```

Table 6
The client program

```
void main (int argc, char* argv[])
{
  .................
  // bind to Search Factory and get its object reference
  SearchFactory_var = ISS_SearchFacto
    ry::_bind(":ISS_YAHOO", hostname);
  //Invoke NewSearch operation to get the object reference of
    Search object
  Yahoo_Search_var = SearchFactory_var->New
    Search( );
  //put query string and invoke AddKeyword operation
  for (i = 1; i < argc; i++){//add query string
   if (argv[i][0] = = '+' || argv[i][0] = = '-'){
    if (argv[i][0] = = '+')
      Keyword_element.Inclu_Exclu = 0x01;
    else
      Keyword_element.Inclu_Exclu = 0x02;
      Keyword_element.item = CORBA_string_alloc(str-
        len(argv[i]-1)+1);
      strncpy(Keyword_element.item, argv[i]+1,str-
        len(argv[i]));
   }else{
    Keyword_element.Inclu_Exclu = 0x00;
    Keyword_element.item = CORBA_string_alloc(str-
      len(argv[i])+1);
    strcpy(Keyword_element.item,argv[i]);
   }
   if (! Yahoo_Search_var->AddKeyword(Keyword_ele-
      ment)){
     ......;
   }
   delete Keyword_element.item;
  }
  //set all of Attributes
  Yahoo_Search_var->Tag(...);
  Yahoo_Search_var->Date(...);
  Yahoo_Search_var->Area(...);
  Yahoo_Search_var->DataBase(...);
  Yahoo_Search_var->DispNum(...);
  //invoke Execute Search operation
  Result_var = Yahoo_Search_var->ExecuteSearch( );
  ISS_Result* result_element;
  // get the result
  CORBA_Long NoResult = Result_var->No_Result();
  if(NoResult! = 0)
   for (i = 1; i < = Result_var->No_Result( ); i++){
    result_element = Result_var->retrieve_element_at(i);
   }
}
```

By this procedure, it is clear that an application wanting to search the Internet neither needs to have a complex search component nor to execute many sophisticated network-accessing efforts. It needs only to issue a few of invocations on the agent.

3. Application of ISS

We build our experimental heterogeneous search engine, i.e. Octopus, to demonstrate the feasibility of ISS. Our system involves two search engine agents, which are implemented as a CORBA object, but using a similar method could allow us to add other search engine agents to the system easily.

3.1. System architecture

Since Octopus utilizes ISS IDL definitions, it provides a single and uniform interface for searching Web documents. On receiving a query from a WWW user, Octopus dispatches it to multiple search engines in parallel, and collates the returned references.

Fig. 4 shows the Octopus system architecture. In this system, a Web user posts a query request via common gateway interface (CGI). The CGI then forks a mediator for each request. The responsibilities of the mediator are as follows: First, it obtains agent information from the Service Repository that stores agent information, such as the number of agents and its name. Then it creates multiple threads to perform this query. In our design, each created thread is a wrapper. Second, it collates the returned information from all of the agents, before merging and filtering them. Finally, it returns the query results to the Web user.

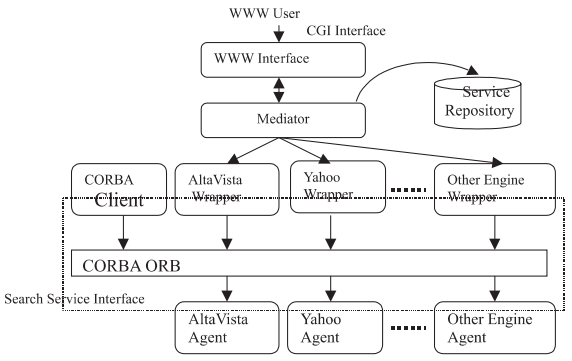

Fig. 4. Octopus system architecture.

In addition, each wrapper is a CORBA client program, which is responsible for setting the attributes for each search engine agent and issuing invocations for operations. Most search engines have their own attributes, although they differ from each other only slightly. Each agent has one wrapper. The first task of a wrapper is to add keywords into the related agent, and set its attributes. Next, it performs the *ExecuteSearch*( ) operation to invoke a search. Once a search engine returns the results to the agent, the wrapper will extract the returned information from the agent, and return it to the mediator. A further clear advantage to our system is that programmers can implement their own wrappers in their applications to perform Web search functions and thus have a search service capability.

Another component in our system is the agent, which is also a CORBA object. When the agent receives the request from a wrapper, it will encapsulate query information into the HTTP format of the related search engine. Finally, it sends the query information to the related agent and obtains the results from the search engine agent. Although we have a uniform interface- ISS, adding other agents into our system is made easy.

Moreover, because these agents are implemented as CORBA objects, a general CORBA client can also use this interface to invoke search engine agents in their application programs. Of course, clients that are developed in other object models, such as Microsoft's COM/DCOM, can also use these agents via CORBA's Internet Inter-ORB Protocol [10] in the same way.

3.2. System implementation

With Octopus, the mediator is an important component. As mentioned above, Octopus serves each query request by a dedicated thread of a mediator. The mediator dispatches the request to multiple search engine agents. Thus the mediator

has to create multiple client threads. Each thread is a client of agent. The codes for creating multiple threads are shown in Table 7.

On the other hand, although no single search service is sufficient in the WWW, and a heterogeneous multi-search service may include most references, many search engines may return duplicated references, and two search engine agents may return the same references, which results in confusing users. In Octopus, duplication is avoided by using a hashing function in the mediator for filtering the references. Any returned reference with the same network address as a previous one is discarded to guarantee its uniqueness. In our experiment, the returned references from Yahoo and AltaVista search engines are 2286 and 4000, respectively for 20 query items, while Octopus has 5183. Clearly, the two search engines duplicated 1103 references, which gives a duplication rate 21%. In addition, Octopus uses a dedicated thread to guarantee the availability of the filtered references.

Another important component of the architecture shown in Fig. 4 is the wrapper. This acts as the client of the search engine agent. The implementation is similar to the CORBA's client described in 2.4.

Another feature that is provided in many search engines is the weight of responses. The weight represents the relevance of the responses to the query string. Each search engine has a proprietary weighting algorithms. In most search engines, query results are shown on the result pages in order of weighting. However, Yahoo and AltaVista responses do not show weights, and so in a MSE it is hard to accurately evaluate the weight of each item. If the search engines do not respond with the weight in a result, the weighting algorithm in Octopus simply gives each item a weight of 500. In Octopus, the weight is implemented by normalizing the scores returned by search engines to between 0 and 1000. Then the mediator calculates the average for all the weights from the search engine agents. We do not attempt to improve the integration beyond this ad hoc approach, because our focus was on proposing modularized architecture and interfaces. We ensure with Octopus that items are shown in order of weighting. It is a simple approach that can be easily changed in the future in modularized architecture. We can also apply other weighting algorithms to Octopus, such as [14].

Table 7
The code segment of a multi-threaded version

```
hAgentTread[0] = (HANDLE)_beginthreadex(NULL,0,
    Yahoo_Client, &user_no,0,&AgentThreadID[0]);
hAgentTread[1] = (HANDLE)_beginthreadex(NULL,0,
    AltaVista_Client, &user_no,0, &AgentThreadID[0]);
WaitForMultipleObjects(2,hAgentTread,true,INFINITE);
```

How to merge search results from search engines is also an important issue in a multi-search engine. Raw results from individual search engine agents must be integrated for display to the user. Results can be displayed with little additional formatting and can be rank ordered or interleaved. In Octopus, the number of items received on one page is 100 from Yahoo and 200 from AltaVista. This is in order to promote the system performance. The maximum number of retrievable items from the two search engines is 200. In Octopus, the agent only fetches the first page (HTML file) that covers the URL and the description of each item. Therefore, the approach to merging all the search results is to filter those that come from the agents and weigh the filtered results. Similarly, the query results are shown in order of weighting.

The mediator has a central role in Octopus. Other valued-add services may also be integrated into the mediator to enhance the system's capability.

3.3. User interface

The underlying systems in Octopus are two platforms (a SPARC and a Windows NT). The ORB of this system is IONA's Orbix 2.02 [15], which fully complies with CORBA specification. The conventional search engines here included in Octopus are two typical search engines – Yahoo and AltaVista search engine. Figs. 5 and 6 show the user interface. When a user submits a query using the query form (Fig. 5), the system will perform the query process and obtain the results from Octopus. The returned results are organized and displayed in a unified form, as shown in Fig. 6.

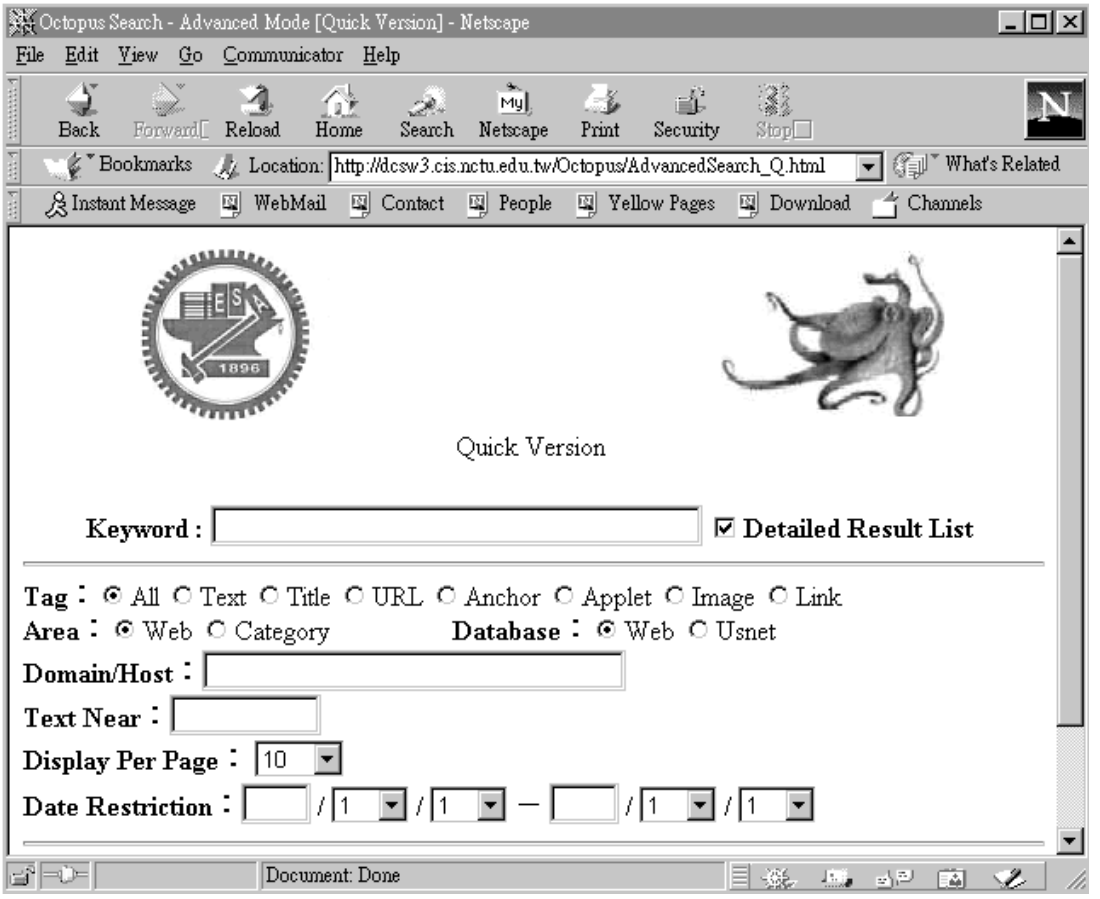

Fig. 5. Octopus user interface.

The user may either issue a simple or a complex search. For a complex search, all of attributes shown in Fig. 5 are adjustable and for a simple search they have a default value.

3.4. Performance evaluation

Performance is very important issue in the client/server model. In Octopus, there is one mediator, multiple wrappers, one ORB, and multiple agents working in parallel in maximize the service. Performance is therefore seriously effected. To improve performance in our system, the following strategies are used: multi-threading and various agents configured on various hosts.

Multi-threaded programming is a well-known technology for improving server performance. In Octopus, the mediator, wrappers and search engine agents are all multi-threaded versions. When a mediator thread is created, it immediately creates multi-threaded wrapper. Each thread of wrapper is associated with one agent that is also a multi-threaded version created by ORB.

In addition, we assign each agent to a dedicated host. This is easily done in the CORBA environment. Thus, the strategy can balance the overhead of the system. We are also aware of object migration [16] techniques to balance the system load for future consideration.

We perform the preliminary measurements shown in Fig. 7 to assess the performance of Octopus and compare it with the Yahoo and AltaVista search engines. Though the overhead of transmission in Internet in most situations is

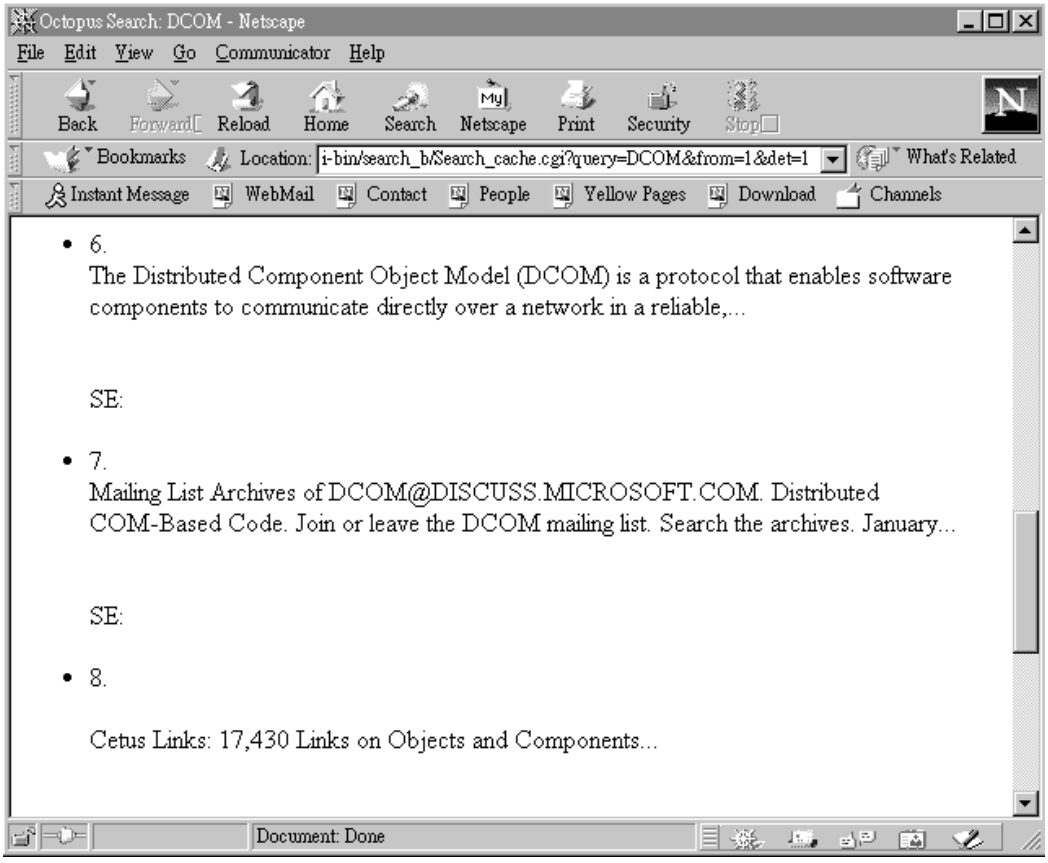

Fig. 6. Octopus returned results.

unpredictable, it is obvious from the figure that Octopus is efficient. The average query time for Octopus is slower than the two representative engines. The reasons for overhead in Octopus are to create multiple threads that execute search operations, to deliver message on ORB, and to filter returned references. As Fig. 7 shows the resulting performance is reasonable. We also measure the total overhead, which as in Fig. 8 shown, is 6.5%. We believe that the major overheads are filtering the returned references and the networking overhead of CORBA. But these operations are executed in parallel, so the overhead does not increase linearly, i.e., it does not rapidly increase with the search engines increase.

Fig. 9 shows a comparison of averaged performance between Octopus and two well-known multi-search engines, MetaCrawler and SavvySearch, respectively. As the returned records of each query request from the SavvySearch are about 70 and each return has a constant number-15, the comparison is limited to a maximum 60. From the results shown in Fig. 9, it is obvious that Octopus is efficient though it has about 0.5 s network overhead that is measured from the test-bed to two real systems. A possible reason for this result is that Octopus is only a prototype multi-search engine based on ISS. It has no applied sophisticated algorithms and many system access operations to handle the information returned from the search engine agents. In addition, evaluating network behavior, such as the numbers of packet retransmission, is difficult.

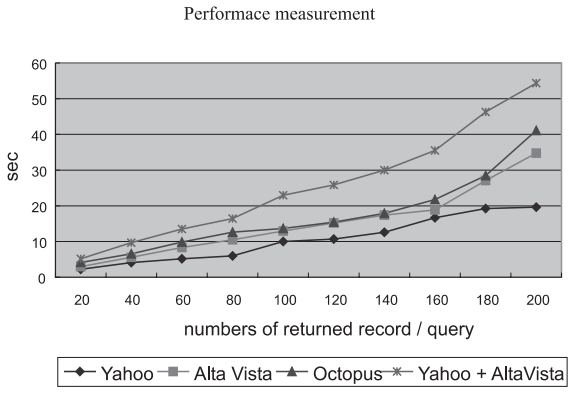

Fig. 7. Performance measurement.

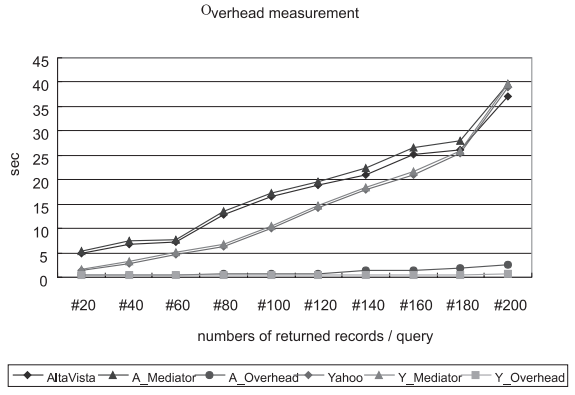

Fig. 8. System overhead measurement.

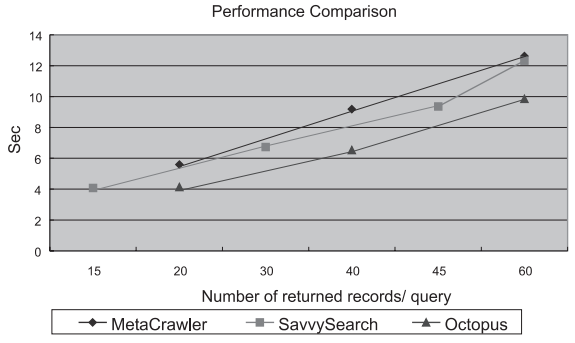

Fig. 9. Performance comparison.

4. Discussions and future works

4.1. Advantages

CORBA is an industrial standard, which supports more than a dozen services in the COSS. There are many benefits to making a large client/server middle-ware based on CORBA [20]. In designing and implementing our ISS-based Octopus, we can mention further advantages.

First, with the progress in search engine technology, more powerful search engines can also be tied into the system in a similar way. The system developer can easily tie new search engines into the system simply by creating search engine agents. In our experience, 80% of all codes found in agents are the same, because the agents have the same server stub that is generated by IDL compiler. To create a new agent, only a small part of a program needs rewriting. All the network operations are hidden from the CORBA's server stub. The developer needs only to handle the interface of the search engine when constructing the new agent. The remainder of the system can be retained.

Second, it is easy for programmers to build applications that need a search ability. Application programmers utilizing the interface to search for information in their application can hide the complexity from network programming and concentrate most effort on other significant value-added services. After the search engine returns the results, the program does not need to extract the information from the complicated HTML file. Since, they are all based on the same interface, applications are undiscerning when querying agents. In addition, through the CORBA standard, applications can also be developed in other environments, such as COM and JAVA. But, applications so developed must be mediated through IIOP bridges.

Third, CORBA is a distributed object-oriented environment. In Octopus, agents can be easily distributed at different locations. In this way, balancing the load while the size of system is on the increase is easy. The system manager can dynamically add other agents into the system. Thus, a system based on ISS naturally has scalability.

Finally, because this is a modularized and component-based approach, it will be easy in the future to replace certain components with new and useful algorithms, such as a weighting algorithm and a natural language processing algorithm. Similarly, a system based on ISS naturally has flexibility.

4.2. Extension and future works

In addition, as described in Section 1, the ISS is easily applied to other types of search engines. For example, many libraries allow user to inquire the book information on the WWW [17,18]. Such a service provides a query on a Z39.50 server to be made via a Z39.50 gateway [19] that translates the query string into Z39.50 format. The scenario is the same as for a general search engine. Thus, integrating such a service into the system can follow a similar procedure.

Users generally query book information from libraries via *Subject*, *Title* (*book or journal*), *Author/s*, *ISBN*, *ISSN*, *or Keywords* from libraries. These styles can be seen as a query attributes. Therefore, the *Tag* attribute in the *Search* Interface of ISS can merge all the attributes because it is declared as *Long* type and the representation is bitwise. All ISS interfaces do not need modification. Before constructing an agent we need only analyze the query string that is sent to the server. The implementation steps and approaches to the agent are same as for the general search engine agent in Octopus. Obviously, this approach can be extended to merge heterogeneous search services.

Octopus is only a prototype multi-search engine that ties with two agents. To increase its usability, we shall construct more search engine agents in the future. We shall also add a few value-added services, such as personalized function that is a popular feature in most search engines.

5. Conclusions

In this paper, we have proposed an Internet search service (ISS) based on the CORBA, which is an industrial standard of a distributed object-oriented platform and has been announced by OMG. We have followed the style of COSS to define the interface of the ISS. In addition, according to ISS interface, we have constructed two representatives of search engine agents – Yahoo agent and AltaVista agent. Since, the ISS is based on the CORBA and was implemented as an Internet search component, programmers can use the interface to search Web site on the Internet in their applications via these two agents. We have integrated these two agents and built a multi-search engine prototype – Octopus. As these agents are implemented as a CORBA object, and the interface is uniform, other search engines are readily tied into Octopus.

The major contributions of our work have been as follows: First, we proposed a uniform interface that accommodates most search engine interfaces. The function of ISS is useful in the general applications that need a search service capability. Programmers can use this interface to construct search engine agents or to query search engines on the Internet in their applications. Second, because the interface is uniform, we can easily build a multi-search engine. In addition, we can easily tie a new search engine into the multi-search engine. Existing multi-search engines do not have this capability.

In our experience, using ISS to implement either search engine agents or multi-search engines is easy. We also believe that ISS can be easily extended to other types of search engine agents, such as knowledge- or data-discovery, real-estate systems and digital libraries.

Acknowledgements

The authors thank the referees for their valuable comments. This work was partially supported by National Science Council of Taiwan ROC under grant NSC88-2213-E-009-087 and by Institute of Information Industrial of Taiwan ROC under grant No. C87-144.

References

[1] E. Selberg, O. Etzioni, Multi-engines search and comparison using the metacrawler, in: Proceedings of the Fourth World Wide Web Conference'95, Boston, MA, 1995.

[2] Chia-Hui Chang, Ching-Chi Hsu, Customizable multi-engine search tool with clustering, in: Proceedings of the Sixth International World Wide Web Conference'97, Santa Clara, CA, April 1997, pp. 257–264.
[3] D. Dreilinger, Integrating heterogeneous WWW search engines, May 1995. ftp://132.239.54.5/savvy/report.ps.gz.
[4] D. Dreilinger, A.E. Howe, Experience with selecting search engines using metasearch, ACM Trans. Information Systems 15 (3) (1997) 195–222.
[5] O. Etzioni, D. Weld, A softbot-based interface to the Internet, Comm. ACM 37 (7) (1994) 72–76.
[6] A. Moukas, P. Maes, Amalthaea: an evolving multi-agent information filtering and discovery system for the WWW, Autonomous Agents and Multi-Agent System 1 (1998) 59–88.
[7] M.A.C.J. Overmeer, A search interface for my questions, Comput. Networks 31 (21) (1999) 2263–2270.
[8] M.A.C.J. Overmeer, My personal search engine, Comput. Networks 31 (21) (1999) 2271–2279.
[9] G. Booch, Object-oriented Design with Applications, Benjamin Cummings, Menlo Park, CA, 1991.
[10] Object Management Group, The Common Object Request Broker (CORBA): Architecture and Specification, vol. 2.2, February, 1998.
[11] K. Brockschmidt, Inside OLE, 2nd ed., Microsoft Press, Redmond, WA, 1995.
[12] Object Management Group, CORBA services: Common Object Services Specification, OMG Document Number 95-3-31, 31 March 1995.
[13] Sun Microsystems, Enterprise JavaBeans to CORBA Mapping, v. 1.0, 23 March 1998.
[14] L. Gravano, H.Garcia-Molina, Merging ranks from heterogeneous Internet sources, Technical Report: AR_300, Stanford University, Stanford, CA.
[15] Orbix Programming's Guide, IONA Technologies, Novermber 1994.
[16] M. Nuttall, A brief survey of systems providing process or object migration facilities, ACM Operating System Reviews 28 (4) (1994) 64–80.
[17] National Library, http://readopac.ncl.edu.tw/z3950/.
[18] National Taiwan University Library, http://tulips.ntu.edu.tw:211/screens/z39menu_chi.html.
[19] Y.-H. Tseng, Z39.50 server based on WWW(II), http://www.lius.fju.edu.tw/~tseng/papers/lacz39.50-2/lacz3950-2.htm.
[20] R. Orfali, D. Harkey, Client/Server Programming with JAVA and CORBA, Wiley, New York, 1997.

Chang Yue-Shan was born on August 4, 1965 in Tainan, Taiwan, Republic of China. He received the B.S. degree in Electronic Technology from National Taiwan Institute of Technology in 1990 and the M.S. degree in Electrical Engineering from the National Cheng Kung University in 1992. Currently, he is a candidate of Ph.D. in Computer and Information Science at National Chiao Tung University. His research interests are in Distributed Systems, Object Oriented Programming, Fault Tolerant, and Internet Technologies.

Shyan-Ming Yuan was born on July 11, 1959 in Mauli, Taiwan, Republic of China. He received the B.S.E.E degree from National Taiwan University in 1981, the M.S. degree in Computer Science from University of Maryland, Baltimore County in 1985, and the Ph.D. degree in Computer Science from University of Maryland, College Park in 1989. Dr. Yuan joined the Electronics Research and Service Organization, Industrial Technology Research Institute as a Research Member in October 1989. Since September 1990, he had been an Associate Professor at the Department of Computer and Information Science, National Chiao Tung University, Hsinchu, Taiwan. He became a Professor in June, 1995. His current research interests include Distributed Objects, Internet Technologies, and Software System Integration. Dr. Yuan is a member of ACM and IEEE.

Win-tsung Lo received the BS and MS degrees in applied mathematics from National Tsing Hua University, Taiwan, Republic of China, and MS and Ph.D. degree in computer science from the University of Maryland. He is now an associate professor of computer science and the director of Computer Center at Tung Hai University, Taiwan, Republic of China. His research interests include architecture of distributed systems, data exchange in heterogeneous environments, and multicast routing in computer networks.