A Testbed for Information Retrieval Research: The Utah Retrieval System Architecture

Lee A. Hollaar
Department of Computer Science
University of Utah
Salt Lake City UT 84112

Abstract

The Utah Retrieval System Architecture provides an excellent testbed for the development and testing of new algorithms or techniques for information retrieval. URSA[tm] is a message-based structure capable of running on a variety of system configurations, ranging from a single mainframe processor to a system distributed across a number of dissimilar processors. It can readily support a variety of specialized backend processors, such as high-speed search engines.

The architecture divides the components of a text retrieval system into two classes: servers and clients. A triple of servers (index, search, and document access) for each database provide the capabilities normally associated with a retrieval system. Possible clients for these servers include a window-based user interface, whose query language can be easily modified, a connection to a mainframe host processor, or AI-based query modification programs that wish to use the database.

Any module in the system can be replaced by a new module using a different algorithm as long as the new module complies with the message formats for that function. In fact, with some care this module switch can occur while the system is running, without affecting the users. A monitor program collects statistics on all system messages, giving information regarding query complexity, processing time for each module, queueing times, and bandwidths between every module.

This paper discusses the background of URSA and its structure, with particular emphasis on the features that make it a good testbed for information retrieval techniques.

Introduction

The typical information retrieval system is based on algorithms or processing techniques particular to that system, such as a novel high-speed character search or a unique means of indexing the stored information. Other portions of the system are designed around that key algorithm, sometimes using the algorithm in special ways through non-standard entry points. While this may yield a system with good performance, it makes it difficult to modify these key algorithms. More importantly, it is almost impossible to try a completely new processing technique without a major rewriting of the system, which can be more time consuming than developing an entirely new system.

This is particularly troublesome to an experimenter who wishes to determine how a particular algorithm or technique performs compared to an established one. In the process of building the test system incorporating the new technique, "improvements" to the query language or other parts of the established system may be made, with the result being a system with enough differences from the base system that it is impossible to accurately determine if any perceived differences in user performance or satisfaction are the result of the new processing technique or if they come from the "minor" improvements.

As part of an ongoing research and development project examining the use of special purpose backend processors to enhance the performance of information retrieval and handling systems [4, 5], it was necessary to implement a complete information retrieval system so that the operation of a backend search machine could be studied in a representative environment. Existing information retrieval systems were either unavailable for the modifications necessary to accommodate a backend processor, because of their proprietary nature, or were unsuitable for easy modification. It became clear that it would be necessary to develop a new testbed system to serve as a base for demonstrating the operation of the hardware search machine, as well as allow the evaluation of other information retrieval techniques and algorithms.

Designing a New System

As originally proposed, the new testbed system would consist of a number of subroutines, such as the index or search routines, with cleanly-defined interfaces to all system subroutines. Alternative implementations of these subroutines could be linked together to form variants of the basic system, differing only in the algorithms used to implement one subroutine. A subroutine could also be replaced by a new module which sends the necessary parameters to a backend processor, such as the hardware search machine. As far as the rest of the system was concerned, that module was simply a new way of implementing the search previously performed by software.

The basic structure of the system was heavily influenced by previous work on the EUREKA [1] retrieval

system project at the University of Illinois at Urbana-Champaign by the project director and the project's consultants. That system ran on a modified PDP-11/40, and supported databases of about 25 megabytes. It used a partially inverted file technique, with the index indicating that a particular document contained the specified term but with no information about where a particular term occurs within the document. Scanning of documents was necessary for queries involving phrases, proximity, or context, although the index did eliminate the searching of documents with no hope of matching the query.

Rather than restrict the operation of the new system to a particular processor or operating system, it was decided to implement the system in the C programming language. Because of its ability to efficiently use many machine features, C can be viewed as a machine-independent assembly language as much as a high-level programming language. Most of the system could be written as portable code, with only certain submodules (such as the ones to access the data stored on disk) written as machine-specific code.

Advanced Workstations

At the same time that development of the new testbed system was starting, the Department of Computer Science at the University of Utah was becoming involved with the new generation of advanced workstations. These differ substantially from conventional terminals, in that they contain a powerful programmable computer and have large displays with excellent resolution. The operating systems supplied with them allow communications on a local area network, permitting a single file server to handle a number of diskless nodes.

Perhaps the feature of these workstations that had the most visible influence of the design of the new retrieval system was the ability to divide the screen into multiple overlapping windows. Each window can be used for a separate function, such as query entry, document display, system control, or word processing. More importantly, information can be moved from one window to another, allowing retrieved text to be incorporated into a new document being developed in a word processing window. The use of a pointing device, such as a mouse, simplifies this data movement.

This ability to conveniently move information between windows also eliminates the need for special commands to save and reissue queries. Interesting queries can be moved into the window corresponding to a file of saved queries, then moved from that window to the query entry window and altered using the window editing functions to reissue them. Specific terms for a query can be moved in from text previously retrieved.

A Message-Based Approach

The use of advanced workstations, rather than a mainframe computer with attached terminals, suggested that a distributable system architecture be used. Rather than construct the system from a collection of subroutines linked together to form a monolithic system, each distinct processing function (index, search, user interface, etc.) exists as a separate process, communicating either within the same workstation or across a local network to other modules.

The calling sequence of a subroutine is replaced in the message-based approach by fixed message formats for passing data. An underlying communications system routes the messages to the specified module, queueing them for processing. For modules running on the same processor, an interprocess communications facility or even a subroutine call can be used as the communications system, providing good performance and low overhead.

If the module initiating the operation (the client) waits for the module performing the operation (the server) to complete before continuing its execution, that client is following the remote procedure call model, which is similar to a conventional subroutine call. If, instead, the client continues processing in parallel with the server, it is using asynchronous communications. While many of the distributed systems proposed follow one or the other of these approaches, an examination of the types of message passing needed for efficient implementation of a retrieval system indicated that both approaches are necessary.

The Utah Retrieval System Architecture

Based on these considerations, an overall architecture for the implementation of distributable information retrieval systems was developed [6]. The Utah Retrieval System Architecture, or URSA^{tm}, consists of a number of client and server modules, communicating using fixed-format messages, as well as the high-level communications protocol. While it was originally designed for information retrieval, other servers can be added to expand the function of the system. For example, a relational database server could be added to provide operations on structured data.

Figure 1 shows the logical structure of the URSA architecture. The communications network forms the backbone of the architecture. At the top left of the network are the primary clients in the system: the user interfaces running in conjunction with the workstation window management system. A user interface client function could also be provided by a mainframe system, controlling a number of conventional terminals and sending messages to the network similar to those generated by a workstation. The primary servers for the information retrieval function are on lower right. For each database in the system, there is a triple of servers: index, search, and document access. The other functions shown in the figure will be discussed later.

There can be a number of different physical implementations of this system architecture. It could be implemented with the clients and servers fully distributed to their own machines, emulating the logical model, or it can be implemented on a single processor, using a multitasking operating system with interprocess communications. A semi-distributed approach, with groups of functions being provided by the same processor (such as one machine handling both search and document access) is also possible. The most common configuration used for demonstrations is for the user interface to run on one workstation, the backend functions on another, and the network and system control functions on a third. A fourth processor is used to control the hardware search function when that enhancement is being used.

Resource Allocation

A key feature of the architecture is the means for establishing communications between clients and servers. This function is carried out by the high-level protocol modules. Whenever a server comes online, the communications system sends a set of messages to a special server, called the resource allocator or name server. The network address

of this name server is the only address that is known a priori by any server or client. The messages passed by a server to the resource allocator consist of its network address and attribute-value pairs describing the services it will perform. The prime attributes are the basic system function a server will perform (index, search, etc.) and the name of the particular database. Secondary attributes may include the type of processing, such as indexing technique or processing speed.

When a client first requires a service, it also communicates with the name server, supplying a set of attribute-value pairs which describe the service desired. The resource allocator compares these against the lists for active servers and, if a match is found, returns the network address of the proper server to the requesting client. If two or more servers match the client's request, the address of the latest server to come online is returned if the servers have identical attribute lists, or a list describing the differences in attributes not identical is returned to the client. In the latter case, the client can select the server that best fills its needs. Finally, if no server satisfies the client's request, an error return is given. The client can either indicate an error has occurred or wait for the desired server to come online.

Error Recovery and Module Substitution. Once the name server has provided the network address of the desired server to the communications routines at the client, it is no longer needed to support that link. However, if the link ever becomes broken, such as might happen if a server crashes or a network failure occurs, the resource allocator is again used as part of the recovery mechanism. First, a new request for the server address is made, to see if the old address is still valid. If it has changed, an attempt is made to re-establish the link at the new address. If the original server is no longer available, the resource allocator is used to determine the address of a replacement server, and a link is made to this module. Only in the event of no replacement being available is an error indicated to the client.

Perhaps one of the more interesting features of this error recovery procedure is the ability to replace one server with another while the system is operating. All that is necessary is to start the replacement server, creating an entry at the name server identical to that of the module to be replaced. Then when the original server is stopped, the recovery procedure will redirect the link from the client to the new server. The client is not aware that a switch has occurred.

Of course, if the server contains state information regarding the session, this will be lost and a possible error may occur unless some special action is taken during the switching of the modules. For this reason, it is desirable that as little state information be stored in a server as possible. In the current implementation, the only server that stores state information between requests is the index, which saves lists of documents matching past queries. A planned modification to the backend structure will have this information stored by a special server, so they can be accessed by any index server.

The User Interface

The most visible part of the system is the user interface. It is the major client for the various servers, and handles the translation of the query into a standard input form for transmission on the network. It takes the result messages from the servers and formats them for display.

As with the overall system, the user interface also consists of a number of separate modules that operate in parallel and can be distributed. The primary server is the window manager, which supervises the overall display for the user interface. Other modules include the query parser, document display, and online help.

Window Management. The window management server provides a uniform display interface for the other modules of the user interface. It provides one or more blocks on the screen for the display of text (and, in the future, graphics) when requested by client. These blocks can

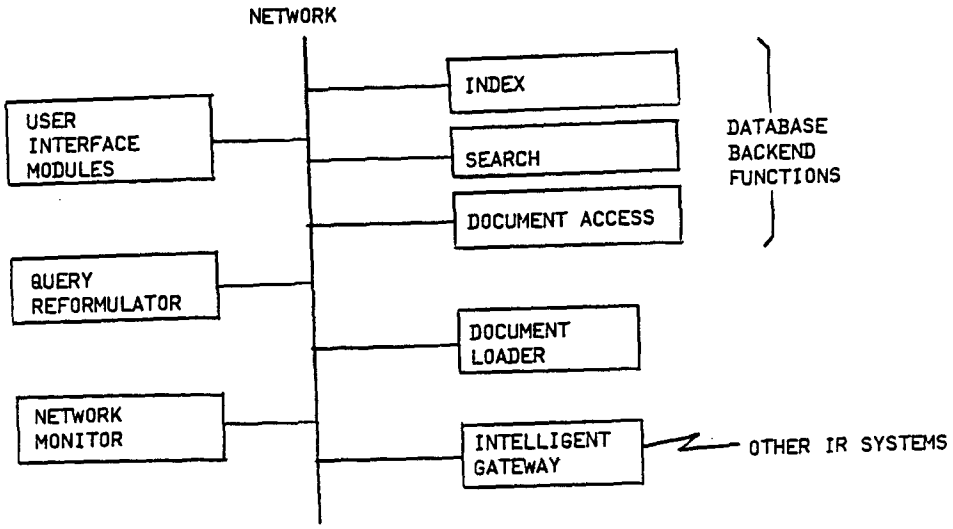

Figure 1: The URSA High-Level Logical System Model

overlap each other, with the active block (the one containing the cursor that indicates where keyboard entries will be made) always in the foreground. The window manager is the only process that directly controls the display screen and the keyboard.

Even though most advanced workstations provide some sort of window management system, the development of a special window management module was necessary for a number of reasons. While the displays produced by most window managers are outwardly similar, the system calls necessary to produce those displays are often quite different. For the user interface to be portable to a wide variety of workstations, all display calls must be isolated in a single module, which can be changed to match the requirements of the particular workstation. In its simplest form, this is service provided by the new window manager: the translation of a common set of display functions to the calls necessary for a particular workstation's window manager.

The window management server also handles the positioning of windows based on a user profile. It provides customizable editor and menu support, so that these functions do not have to be included in each client program. The editor provides the basic cursor control within the window and handles the movement of information from one window to another. The functions of the editor can be tailored for a particular class of windows by providing special handlers. For example, the keystroke command "move to next paragraph" can move the cursor to the line following a blank line in a word processing window, the start of the next query in a transcript window of past queries, or initiate a network request for the next block of text for a document display window.

The inclusion of the editor functions as appendages to the window management server substantially improve their performance over having them in each client, because network communications is not necessary. It also promotes a more uniform set of editor commands across different types of windows.

The menu system also acts as a special appendage to the window server. It is activated when a special mouse button is pressed or other specified event occurs. Three types of menus are supported. Permanent menus are always displayed, unless another window overlaps them, and are used primarily for invoking special system functions, such as starting a word processing window. Fixed-position and dynamic pop-up menus are displayed when the mouse button is pushed, and cleared when the desired menu item has been selected. In all cases, the menu system operates on a special data structure specified for a client program, returning a specified string as if it had been entered through the keyboard to the client when a menu item has been selected. This means that client programs do not need to be modified to take advantage of the menu system. All that is necessary is the building of the proper data structures for inclusion in the menu system.

Query Handling. As would be expected for a retrieval system, the primary clients of the window server are the query parser and the document displayer. When a query has been entered, it is sent to the parser module over the user interface subnetwork, where it is converted to a standard tree representation for transmission to the backend servers. The parsing routines used in this conversion are written using Lex [8] and Yacc [7], the compiler-compiler system from Unix, so that the query language can be easily modified. In fact, less than a week's effort was necessary to replace the normal system query language with an extended version of the LEXIS query language [3].

If the user wishes to see the results of query, a document display process is started and given its own window. From that point on, the document display function is separate from query processing. A new query can be issued without affecting the display of previous results.

Other Functions. A number of additional user interface functions are available, and others can be readily added. Word processing is handled simply by using the appropriate editor routines in the window server. An online help facility exists, operating on a specially formatted version of the system documentation stored in a private database. It uses the conventional index, search, and document access functions to find descriptions on system functions. Because the help information is displayed in a separate window and can be browsed while using the other windows, it is possible to include sample query sessions in the documentation that the user can try, either by typing the queries into the appropriate window or using the mouse to move them there.

Other user interface modules can be provided to handle electronic mail between users, formatting and display of system or user statistics, control of special server functions, or any other desired activity. Because of the modular, distributable structure of the user interface, mirroring that of the overall system, the inclusion of a new user function does not affect any other modules.

Backend Functions

The backend functions are what makes the overall architecture an information retrieval system. For each database accessible by the user, there are three backend functions necessary: an index, a search module, and a document access handler. While the logical structure of the system has three different servers for each database, there is no reason that these servers cannot be combined on a single machine, or in a single program, either by having a single server handle the same function for multiple databases, or by having a server provide more than one function for a single database (such as a combined search and document access program).

The Index. The first server that handles a query is the index. It takes the tree representation of the query and produces two lists of documents. The hit list contains all documents which the index determines match the query based on information known to the index. The maybe list contains those documents with a chance of matching the query, but which must be searched to determine if they should be added to the hit list.

Different indexing schemes result in different information being placed in hit and maybe lists. For example, a very complete index would probably place all documents that match a query on the hit list, leaving the maybe list empty. No index at all results in an empty hit list, and a maybe list consisting of all documents in the system for the first query or the final hit list from a previous query for a subsequent query. A partially inverted file causes entries to be placed on both the hit and maybe lists.

Searching. Documents on the maybe list must be examined to determine if they should be added to the hit list or can be discarded. The search program is passed the same parse tree as was passed to the index, along with the maybe list. When a partial inversion is used, it is responsible for handling context and proximity operations.

Document Access. After a final hit list has been developed, the user can request the display of the documents matched by the query. The document access server cooperates with the document display module in the user interface by extracting the desired portions of a specified document and sending them over the network. It also contains information regarding the context names present within the documents in its database, supplying the names to modules like the query parser.

While it seems that the document access server could be combined with the search server (and, in many implementation, they would be), the architecture keeps them separate because of the different functions provided by each. This allows a centralized search facility to be coupled with decentralized storage of the documents to be displayed. It also can be used to provide an online backup for the database, by having redundant documents.

Other System Functions

Figure 1 also shows a number of other system functions that are not necessary for the basic system's operation, but can provide important enhancements to its capabilities. The document loader is used to provide new information to the index, search, and document access servers for online inclusion into their databases. The intelligent gateway appears to the user interface as if it were the conventional triple of database servers, but translates the queries and forwards them to another information retrieval system, such as one of the commercial offerings. The results from the other system are then reformatted to look like the standard results from the document access server. The gateway allows users to access other information retrieval systems without the necessity of learning a new query language or other commands.

On the left of the figure is a module used to alter a query to improve either performance or precision-recall. It intercepts the query between the user interface and the backend functions, alters it as necessary, and sends it on to the backend servers. This reformulation can be performed based on past queries, user profiles, or other AI-type techniques. This module acts as both a client (of the backend servers, both by passing on the modified query to those servers and by used them during the reformulation process) and a server (to the user interface). By opening connections to the backend servers first, then indicating to the resource allocator that it is a replacement for those servers, it can intercept all queries from a particular user interface.

Testbed Features

In addition to its modular structure, allowing the easy replacement of system functions with ones implementing different algorithms, data structures, or processing techniques, and the flexibility in handling new and different query languages, there are a number of system features that are specifically provided to aid in experimentation or system performance measurement.

Message Monitoring

While the system is operating, a log of all messages can be collected for later examination. Each message is timestamped when it is issued, when it is placed in the server's work queue, when the server starts to process it, and when the result is returned to the client. Using these timestamps, an analysis program determines the processing times or throughput of the servers, the network bandwidth between each pair of modules, and the distribution of message types over time. Special analysis programs can look inside certain messages to determine query complexity, such as the number of terms and the types of operators specified.

A major difficulty with timestamping messages in a distributed system is assuring that the timestamps are all based on the same clock. In particular, the local clock at each node in the system cannot be used without correction. Without this correction, it is possible that the timestamps might indicate that the result of a message was produced by the server before the client even sent it! A special protocol, unseen by the user or by most system modules, exchanges time information between the various nodes. By determining the expected network delays and sending the proper synchronization messages, the time server is able to keep the clocks on all nodes within about one millisecond of each other, sufficiently accurate for message logging.

Calibrated Delays

The calibrated delay module operates much as the query reformulator, intercepting a query before it goes to a server module, although it does not alter the query. By delaying the messages between a client and a server for a given period of time, the apparent response or processing time for a server can be increased, simulating the performance of a different algorithm or a larger database. This time delay can be determined, for instance, by examining the number of terms or the types of operators in a query as the message passes through the delay module.

Program Development

The current implementation includes approximately 80,000 lines of C source code (including comments and blank lines included for formatting) for a single version of the system. Multiple versions of each module exist as new features are added, so that approximately 500,000 lines of code are currently being maintained. All these programs were written during the last 15 months by a group of ten graduate and undergraduate students.

Because the system is not a monolithic program, but is dynamically constructed as different modules are started or terminated through the use of the resource allocator, conventional software management systems, such as Make [2], cannot adequately handle version control. Perhaps the most typical problem is when the developer of a server is working on a new version at the same time another programmer is working a client of that server. If a problem shows up in response to certain actions from the server, it is important that the server not change while that problem is being diagnosed and fixed by the developer of the client. It is also important that the messages to the server from the client not change during testing of the server for a given query. Another interesting case is when underlying support programs, such as the communications protocol, used by most programs, are changed.

Subneting

The solution is to allow the establishment of logical subnets for different developers, allowing them to run their own versions of the system modules unaffected by other users or developers. When a new version of a module is ready, it can be added to the normal system library.

A logical subnet is developed by selecting a particular version of each system function, as well as underlying modules like the communications system and the time server, and building a temporary library containing them. Also included in the library is information on how to reach a copy of the resource allocator specific to the logical subnet. As servers are started, they communicate with the special resource allocator rather than the normal system name server, so that connections are only established to modules in the developer's private library.

It is common to see two or more programmers working on system development, each using their own subnets for program development and testing, possibly at the same time the demonstration version of the system is running. In fact, it is possible that two programmers could be working on the system at the same time, each with portions of their development system running on the other programmer's workstation.

System Status

A prototype of the URSA architecture has been operational since late 1983, and has demonstrated that the claims made for the architecture hold. Because it was just a demonstration prototype, little effort was made to make it efficient or portable. To the extent possible, Apollo routines such as their window management system and communications techniques were used. The database used was the architectural design document, consisting of about 30 different sections, each of one to five pages. It took less than three months from the start of programming until the prototype was demonstrated to visitors.

After testing of the prototype, development of a new implementation was started. That version is now operational. It uses new communications modules to permit operation across a variety of networks, including the ARPA internet. New servers, capable of handling medium-scale (about 25 megabyte) databases have been included. The currently limitation is imposed by the access overhead of the Apollo filing system for large random-access files. It will be removed when a special-purpose filing system is implemented. The portable window manager is now operational on the Apollos, as well as Sun workstations.

Future Plans

Versions for systems other than the Apollo are also planned. The backend functions will be ported to a VAX/Unix system, and the entire system, including the window management system and user interface, to the Sun workstation. Since all these processors are connected by the Computer Science Department's Ethernet, the system will be able to distributed in any arbitrary way across the different machines.

A special version of the window management server will also be developed for the IBM Personal Computer. Rather than provide overlapping windows, because of the small screen resolution a means of rapidly changing screens will be used. Some initial tests have shown that if window switches can be made almost instantaneously, the convenience is similar to a more elaborate window system. The preliminary version will have only the window management functions running on the PC, with the rest of the user interface (such as the query parser) running on one of the other machines on the network. A later version will allow the complete user interface to run on the PC, connected to the backend over either an Ethernet or a high-speed serial link.

Arrangements are now underway for the testing of the system at two or three locations. This testing should start this summer, after the training of system personnel at each of these locations, and take six to nine months. Before the end of this test period, the decisions about how the system will be distributed, and the cost of any licenses and maintenance charges, will be made.

Acknowledgments

The design and implementation of any system of this magnitude is the work of a number of people. In addition to the author, who is the project director and lead designer, the architecture was developed by Shane Robison and Michael Zeleznik. Mike has also been responsible for the communications protocol design and implementation, in addition to being the assistant project director. The implementation staff included Bob Elens, Steve Voelker, Ellen Gibson, Ian Elliott, Jim Schimpf, Dave Schlegel, Koah-Hsing Wang, Brad Hutchings, and Kurtis Bleeker. Roger Haskin and Perry Emrath were consultants on the system design.

References

1. T G Burket and P E Emrath. User's Guide to EUREKA and EURUP. 79-956, Univ. of Illinois Dept. of Computer Science, Feb., 1979.

2. S I Feldman. Make - A Program for Maintaining Computer Programs. Bell Laboratories, Aug., 1978.

3. E S Gibson. An Extensible and Flexible Query Language for an Information Retrieval System. Master Th., Univ. of Utah Dept. of Computer Science,Aug. 1984.

4. R L Haskin and L A Hollaar. "Operational Characteristics of a Hardware-based Pattern Matcher". ACM Trans. on Database Systems 8, 1 (March 1983).

5. L A Hollaar. "The Utah Text Retrieval Project". Information Technology: Research and Development 2, 4 (Oct. 1983), 155-168.

6. L A Hollaar (ed). The Design of an Extensible Communications-Based Full Text Information Retrieval System. Univ. of Utah Dept. of Computer Science, March, 1984.

7. S C Johnson. Yacc: Yet Another Compiler-Compiler. Bell Laboratories, July, 1978.

8. M E Lesk. Lex - A Lexical Analyzer Generator. Bell Laboratories, Oct., 1975.