Structural Components of the Isite Information System

Kevin Gamiel (Kevin.Gamiel@cnidr.org)
Nassib Nassar (nrn@cnidr.org)
Clearinghouse for Networked Information Discovery and Retrieval (CNIDR)

Abstract

This paper discusses various technical topics related to the structural model of Isite, an open information system supporting multiple external protocol-based access mecha nisms, a database-independent search and retrieval API, and an extensible field-based text search engine. Isite was developed by the Clearinghouse for Networked Information Discovery and Retrieval (CNIDR), which is funded by the National Science Foundation.

Two Models in Data Publishing

The broad range of existing distributed data access models can be classified into two main groups: those oriented toward browsing, and those supporting searching capabilities.

Examples of browsing systems are Gopher (University of Minnesota) and World-Wide Web (CERN); publishers with the proper tools can make their data accessible to users through these systems. Browsing systems are based on the concept of navigating through a virtual information space. Such a system is well suited to data retrieval in cases where there are clear relationships among the data. In such systems, it is very important to have some method of organization, since browsing depends on the user's ability to make intelligent navigational decisions. Browsing systems have proved very useful for locating databases on the vastly distributed global Internet, because the "hyper-text" model utilized by Gopher and the World-Wide Web has encouraged a rudimentary kind of organization among related documents.

An information system based on the searching model is desirable when the data being published are difficult to organize. In such cases an automated search on relevant words of text may be far more economical than a browsing model, or at least may assist the browsing system at certain stages.

In practice, a combination of these access models is frequently used. For example, many search-based systems are arranged at the top level with a user interface provided by a browsing system. It is also desirable that the various searching systems conform to Internet protocol standards so that they may easily be integrated with other searching and browsing systems.

Isite

Isite uses the combined model, but also provides multiple access mechanisms via standard protocols. Additionally, using the Search API included in Isite, multiple databases of different formats may be served. Newly installed data bases have the immediate benefit of being accessible through all data retrieval communications protocols supported by Isite.

At present Isite supports four access methods based on standard Internet protocols: Z39.50, World-Wide Web (HTTP), Electronic Mail, and Gopher. The Z39.50 protocol, implemented in Isite in conformance to the ANSI/-NISO Z39.50 version 2 standard, is the primary access point of the system. Z39.50 supplies a rich set of "stateful" services for deep search and retrieval on distributed database servers. It is the internationally accepted open standard for search and retrieval over networks. The other protocols are supported via protocol "gateway" soft ware. A gateway is software that translates between two different protocols. Thus Isite contains an HTTP to Z39.-50 gateway, which converts search queries received by an HTTP server to queries that can be understood by the Z39.50 server.

The Search API (SAPI) provides a layer of abstraction between the Z39.50 server in Isite and the database system that contains the data to be served. The API is a specification for a certain set of functions that must be supported per database system. Thus the Z39.50 server can rely on a consistent communications layer for accessing data stored in various databases. A supported database may be a complex text search or relational system, or it may be a simple utility such as "grep." The essential principle is that the SAPI must normalize the behavior of the database software in order to be accessible by the Z39.50 server. With this layer operational, all access methods available to Isite are extended to databases supported by SAPI.

Implementation

Isite consists of several distinct software packages, including (1) libcnidr, a source code library of commonly used functions, (2) ZDist, an ANSI/NISO Z39.50 version 2 programmer's library, UNIX server, and UNIX client and gateway, (3) SAPI, the Search API, and (4) Isearch, a field-based text search system including a C++ class interface and Iindex/Isearch utilities.

ZDist

The ZDist package contains much of the core of communications access for Isite. Access to SAPI-supported data bases is entirely directed through Z39.50. The current version of ZDist, included in Isite, supports the Initialize, Search, Present, and Close facilities of Z39.50.

The Z39.50 library is written in C and is based on the freely available BER utilities developed by the OCLC On line Computer Library Center, Inc. The server and client/gateway use this library to encode and decode Z39.50 Protocol Data Units (PDU). The application-level network communications source code is distinct from the Z39.50 PDU encoding/decoding layer, and is physically located in the libcnidr library.

The Z39.50 server application is easily configurable by modifying a text file, and the configuration options include executing the server either as a forking daemon or from inetd, specifying a maximum number of simultaneous connections, specifying which databases to serve, and various Z39.50 settings down to the PDU level. The server's overall operation is straightforward: it (1) listens for client connections on a well-known port (usually 210), (2) accepts a connection, initializes with the client, (3) accepts search queries, and (4) interacts with the SAPI to process the query and returns results to the client, after which (5) the client may request that the server "present" the contents of any of the results, and (6) the server contacts the SAPI to retrieve those contents and return them to the client.

The Z39.50 client is designed especially to be the foundation for gateway applications. Thus it is not interactive, nor does it provide a user-friendly interface. In order to build a gateway from a "stateless" protocol such as HTTP to a "stateful" protocol such as Z39.50, state requirements must be hidden from the calling (in this case, HTTP) side. The Z39.50 client is implemented as a single-pass client that initializes, searches, and presents in uninterrupted sequence. The client also conforms to the Common Gateway Interface (CGI), a well-known standard for gateway communication with HTTP servers. The CGI process provides a layer between HTTP and the Z39.50 gateway. Like the Z39.50 server, the client is configurable, allowing the user to specify all Z39.50 PDU-level information, for maximum flexibility.

Electronic Mail Gateway

Implementation of an Electronic Mail gateway is simplified by taking advantage of a feature of the Z39.50 client. Since all PDU-level information may be specified in a text configuration file, a rudimentary gateway can use the exact text of an Electronic Mail message as the client configuration file. The results of the search may then be mailed in reply to the requesting user.

Isearch and the Document Type Model

Isearch is designed as a set of modular components, at the center of which are two groups of C++ classes: the Isearch engine, which encapsulates the functionality of field-based indexing, searching, and presenting, and the Document Type class hierarchy, which defines the behavior of the Isearch engine for certain types of documents. It is the latter that we wish to discuss here.

Document Type classes bind specific functionality in the Isearch engine to the documents being processed. C++ classes defining the field structure and presentation characteristics of certain types of documents are therefore grouped by the common features of those documents. The DOCTYPE base class defines default behavior, and it is invoked during the processing of documents for which no Document Type class has been specified. All Document Types must be derived from DOCTYPE; therefore it is important to understand the implementation of the base class and its interactions with the Isearch engine, which, although straightforward, require some explanation.

There are four essential methods of DOCTYPE: AddFieldDefs(), ParseFields(), ParseRecords(), and Present().

The first three of these are invoked during indexing, and the fourth during searching. AddFieldDefs() provides information to the Isearch engine about the fields it expects to discover during the parsing phase. ParseFields() is called during indexing of each document record, at which time it parses the document and inserts field structure information into the RECORD object. ParseRecords() defines the record structure for files that contain multiple document records; this may be beneficial in cases where record structure is dependent upon other aspects of document structure that must be determined at run-time. Finally, the Present() method defines exactly how documents are displayed in response to requests for various element sets, which allows presentation to be abstracted from the retrieval of field contents.

The Isearch engine calls each of these methods at appropriate times during the indexing and searching processes. In DOCTYPE they are defined for minimal functionality, but they can be overridden within descendent classes. For example, DOCTYPE::AddFieldDefs() and DOCTYPE::ParseFields() contain no source code, and consequently the default behavior of the indexing routines is to treat documents as lacking field structure. No knowledge of field structure is indigenous to the Isearch engine. However, source code to handle field-based searching is present in the engine and has only to be enabled by defining field structure within the Document Type.

The method of building field definitions and structure tables is slightly involved because of the various levels of nesting and data hiding.

An instance of the DFDT (Data Field Definitions Table) class is stored within the Isearch engine classes, and DOC TYPE methods can add new definitions to the table. The table is a list of DFD (Data Field Definition) objects. Each DFD consists of a field name and an attribute list (ATTRLIST), which in turn contains a list of attribute (ATTR) objects that can be used to preserve additional information about each field definition. Field definition information can be added to the engine's internal DFDT object at any point within the Document Type methods, except that field structure information should not be generated for a record if that field structure refers to fields that have not yet been defined. There are two good reasons to generate DFD objects within DOCTYPE::AddFieldDefs(); first, it is a convenient place to collect all field definition information, and secondly, it allows for optimization within the Isearch engine. An example in which it is necessary to generate field definitions in DOCTYPE::ParseFields() is when field information is not known ahead of time, such as in the case where the field name must be derived from contents of the document being processed.

An example that illustrates this last point is the SGMLTAG Document Type, which scans the document text for SGML-like tags and treats the tag name as the field name and the delimited text as the field contents. In order to support maximum variety of field structure and yet avoid the unnecessary overhead of definitions for fields not present in any of the documents, generating field definitions in AddFieldDefs() would require knowledge of field information *a posteriori*. The problem is solved by adding each field definition as it is discovered during parsing of the document records.

Building field structure information is similar to building a DFDT. Field structure is encapsulated in a DFT (Data Field Table) object, which is a member of the RECORD class; thus there is one DFT per document record being indexed. The DFT is a list of DF (Data Field) objects, each of which consists of a field name and an FCT (Field Coordinate Table). The FCT is a list of coordinate pairs (FC) that delimit instances of the field within the document record. Inserting multiple FC objects into the FCT enables support in the Isearch engine for repeating fields, which may be defined as a sequence of multiple occurrences of the same field within a single document record.

At search time these field definitions and structures may be retrieved and used for presentation of document text. The Isearch engine provides a method for locating the contents of a certain field within a certain document record. However, the Document Type architecture creates one additional level of abstraction. A method in the Isearch engine called Present() is accessible to the main application for general purpose high-level presentation of document text, and it yields control to the Present() method of the Document Type associated with the document record that is being accessed. The default behavior, defined in DOCTYPE::Present(), is more or less to treat the element set as a field name, and to retrieve the contents of that field from the Isearch engine. In addition, it interprets the element sets “B” and “F” as requesting “brief” and “full” records, respectively. The purpose of this architecture is to allow DOCTYPE::Present() to be redefined in descendent classes, in order that various forms of presentation may be implemented. For example, element sets may be synthesized from more than one field, retrieved text data may be reformatted for suitable output, etc. DOCTYPE::Present() essentially “intercepts” normal processing of field-based presentation, thus allowing it to be extended in relation to the document being accessed.

It is hoped that the advantages of using the Document Type architecture outweigh the small amount of additional development required to integrate it with existing systems. Since the model is a map of document type behavior rather than a physical document representation, the scope of the representation is unspecified. A Document Type may be so general as to handle all SGML documents, or so special as to be tailored to a particular proprietary document format. The model may even be wrapped around other field parsers, since a Document Type may be defined simply to read field coordinates out of an ancillary file.

The Document Type model benefits from some of the features of object-oriented programming, including extensibility, modularity, code maintainability, data hiding, and the ability to build upon previous work via class inheritance. Documents defined by a Document Type know how to “present” themselves, which minimizes risky and tedious internal modifications to the Isearch engine. In addition, DOCTYPE can be expanded to provide increased access to features of the engine, while descendant classes may add document type-specific functionality.

Since the Isearch engine supports document records of different Document Types within the same database, it is possible to abstract multitype text data by normalizing functionality within the Document Type class methods. For example, a database of international patents may consist of a variety of data formats. Rather than requiring massive data conversion, the Document Type model allows run-time format normalization by supporting customized field parsing and a formatted presentation layer. Thus multitype data can be stored in their native format within the same database, and Search and Present operations can be abstracted from structural differences.

Conclusion

Isite implements a variety of modular architectural structures based, where possible, upon open standards. Used together, they provide a powerful, extensible information system that is capable of remaining compatible with continuously changing paradigms.