Martin Mareš

mj@atrey.karlin.mff.cuni.cz

PRESENTS

Sherlock

A NETWORK DOCUMENT SPACE SEARCH SYSTEM

VERSION 1.0

1. Introduction

Sherlock is an universal extensible system for collecting documents distributed across the network (e.g., on the World-Wide Web), indexing them and offering full-text search capabilities.

The system is divided to two major parts: the gatherer and the search engine. The gatherer is responsible for walking the network, collecting all data, analysing them and put all extracted information to the object database. The search engine (which can be running on a different computer) creates a searchable (i.e., indexed) database and does the requested searches.

2. Runtime Directory Structure

The Sherlock system stores all its data in a relatively simple directory structure for storage of all its data. Few subdirectories are used to distinguish between different data types and stages of object processing:

`bin/`	All programs in their binary form
`cf/`	Configuration files for all system components
`db/`	Database files and queues used by the gatherer
`host/`	Host description objects (256 subdirectories corresponding to first two digits of host ID)
`log/`	Log files
`obj/`	Document objects (256 subdirectories corresponding to first two digits of object ID)
`sdb/`	Search engine databases
`tmp/`	Temporary files and a queue of downloaded objects. All temporary files in this directory have a unique naming scheme: `Xyyyyyyyy`, where `X` is file type (`t`=generic temporary file, `c`=queue control file (object description directly after downloading), `d`=queue data file (data files referenced by `@` fields in `c` files), `x`=object data after decoding, `s`=object data after charset transformation, `o`=object data after distilling) and the `y`'s form a pseudo-random number.

3. The Object Interchange Format

Each database object usually corresponds to one gathered document and it's stored as a sequence of text lines (usually in a separate file). Each line contains single database field and starts with a control character (usually a letter) specifying field type. Each object is uniquely identified by its object ID (a 32-bit integer, codes less than `0x10000` are reserved for special purposes).

All database fields are enumerated in the following table. Each field is described by four characteristics:

* Field type (fields that can occur more than once for a single object are marked with a dagger (†) sign)
* Value type (`S`=string, `I`=unsigned 32-bit integer, `X`=unsigned 128-bit integer expressed in hexadecimal, `M`=message starting with 4-digit error code followed by a space, `D`=datestamp (number of seconds since the default system epoch), `U`=URL, `O`=object ID (hexadecimal 32-bit integer))

* List of allowed contexts where this field can occur (empty=any, D=in downloader output, C=in distiller output, O=in standard object tree (i.e., on gather daemon output), S=in search engine object pools)
* Field name

U†	U		URL of document described by this object. If there are multiple U fields, the object occurs identically on several places.
B	B	DCO	Base URL for all relative references stored in the document. This field can occur during the whole gathering process, but is usually used only by the distiller.
T	S		Content-type (in D context including a charset, which is moved to the c field later)
E	S	D	Content-encoding (before the object is decoded; moved to n field afterwards)
n	S	COS	Original content-encoding (after the object is decoded)
L	D		The time of last modification of the object
D	D		The time of last download (or modification check) of the object
R†	U		Reference to another URL (all modules involved in the gathering process (downloaders, decoders and distillers) can generate all reference fields whenever appropriate)
J†	U		Java applet code reference (semantics same as for R)
j†	U		Java applet class reference (semantics same as for R)
f†	U		Form reference (semantics same as for R)
I†	U		Image reference (semantics same as for R)
F†	S	DC	Reference to another object file (if any module involved in the gathering process extracts another object from the one being processed (for example when unpacking an archive), the additional objects are moved to temporary files and referenced by this attribute in the original object)
C	S	COS	First few characters (at most 256) of document contents
H	S	COS	Document heading
W†	S	COS	List of words contained in the document. Each word should occur at most once in *all* W fields and can be prefixed by a control character expressing a quality factor ranging from 0x80 (very bad) to 0x9F (very good).
M	X	COS	MD5 sum of the document (after it's decoded and translated to the canonical character set)
N	S		Base name of the object (after extension removal when decoding or name inside an expanded archive)
O	O	S	Reference to original object ID in case object ID's were reindexed (which is exactly what happens in the search engine object pool)
S	S		Server type (name of server software detected when downloading this document)
P†	S		If the document was downloaded via a proxy, this holds the name of that proxy
l	S		List of languages contained in that document (if the server has told us by a MIME header)
e	D		Date and time of expiry (if advertised by the server)
.†	S		General note (for example a warning about object data being truncated)
x	I		Number of type conversions (to prevent looping)
c	S		Name of original charset (extracted from the C field or guessed)
s	I		Document size (when downloading via the first specified URL)
-	M	DC	Document is invalid (includes reason)
X	M	DC	Document has no contents (maybe only references), don't store it to the object pool, only mark as known (includes reason)
0		D	Not modified (as a reply to if-modified-since requests)
?	M	D	Temporary error (it has sense to retry later)
r	I	D	Retry after given number of seconds (can be specified in case of temporary errors)
v	M	D	Use older version of HTTP
+	M	D	Downloaded OK
@†	S	DC	Name of file holding raw object data (if there were any conversions, the most recent form of the file is present in the *first* @ field)

3.1. Host Information Objects

There is a special form of the object files that describe known *host-port-protocol* triplets instead of documents.

These objects contain a completely different set of fields:

`H`†	`S`	Host name (can occur multiple times if the host has some aliases)
`P`	`I`	Port number (usually a TCP port as most information retrieval protocols are based on the TCP)
`p`	`I`	Protocol number
`L`	`D`	Date and time of last access to the server
`f`	`S`	Access filter extracted from the `robots.txt` file on that site (list of disallowed path prefixes, each one present in a separate `f` attribute)
`F`	`S`	Manually supplied access filter (in the same format)
`R`	`D`	Datestamp of the last attempt to download the `robots.txt` file
`S`	`S`	Server software name (updated from the objects being downloaded)
`A`	`I`	Total number of successful accesses
`E`	`I`	Total number of accesses ending with a permanent error
`e`	`I`	Total number of accesses ending with a temporary error
`O`	`D`	Out of service until specified date and time
`C`	`I`	Number of temporary errors when downloading the last URL
`-`	`M`	Last error message issued by the downloader
`D`	`I`	Manual override of minimal inter-access time
`r`	`I`	Manual override of robot file timeout
`d`	`D`	Use older version of protocol and try again the new one after specified date/time.

4. The Gatherer

The gatherer runs as a background daemon, maintains the queue of URL's to be processed, calls downloaders to download documents pointed to by the URL's, converts the documents to an canonical format (i.e., decompresses them, recodes charsets etc.), runs distillers (the programs that extract useful information such as word lists, headings and references from the documents) and finally adds the results to the object pool and also back to the queue.

4.1. Gatherer Data

The gatherer daemon uses the following data structures:

- The URL database (implemented as a hashed file using the `GDBM` library) – maps each URL the gatherer has seen to (o, t, q) triplets where
  - ∗ o is an object ID in case the object has been already downloaded, an error code (numbers less than `0x10000` not corresponding to any valid object ID) in case the URL is invalid, 8888 if the object has no contents (e.g., if it's only a redirect) or 0 if we haven't seen the data yet.
  - ∗ t is the time of last object operation (of last download if $o \neq 0$ or of adding to the database in all other cases).
  - ∗ q is 1 iff the object is queued in the gather daemon's URL queue.
- The MD5 database (also a `GDBM` hashed file) – maps 128-bit MD5 fingerprints of object data to their object ID's. Used to determine whether two URL's point to the same document or not. The probability of two different documents having the same MD5 sum is about 10^{-40} and if Murphy will smile at us and we find them, we will contribute a lot to the cryptographic research…
- The URL queue containing URL's of all documents to be downloaded (marked with $q = 1$ in the URL database). As it's desirable to switch hosts as freqently as we can (in order to reduce their load without any unnecessary waits), we implement it as a queue of *host-port-protocol* triplets, each triplet accompanied by a host object and a local document queue of URL's relative to that point.
- The object queue containing downloaded, but unprocessed objects. We try to keep the number of objects in this queue reasonably small and don't start downloader processes if it exceeds a user-defined limit. Each object in the queue is represented by an object descriptor (a file holding all object fields as described above) and few (typically one) data files.
- Intermediate objects – temporary forms of yet unprocessed objects.
- The object pool – all processed objects ready for transport to the search engine. The 'processed' state doesn't mean we can't alter them anymore (extra URL's can be added later if we discover another name for the same document).

4.2. The Gatherer Daemon

The gatherer daemon (`gatherd`) runs several processes which accomplish various phases of object processing: typically at most N (a user-defined parameter) downloaders for queued hosts and one distiller. The processes are handled in the following way:

- If there's less than N running downloaders and the unprocessed object queue is not full, get a host (if there's any) from the main queue and start a new downloader for it, checking minimal inter-access or inter-failed-access times as needed. If we have no robot access file downloaded or only an ancient one, download the robot access file instead of the URL.
- If a downloader process has finished, fetch the resulting object. If it reports a temporary error, re-queue the URL for retrying. If it's a permanent error, mark the URL as bad. If the downloading was successful, start object processing:
  - If the object has no contents, treat it as already distilled.
  - Determine content type and encoding (discussed later)
  - If the object is encoded, decode it and repeat the previous step.
  - Determine character set (discussed later)
  - In case it is not in our local character set (currently ISO-8859-2), recode it to that encoding.
  - In case there were any charset-dependent `<META>` tags in the document, remove them.
  - Calculate a MD5 fingerprint and find it in the database. If already known, append a new `U` field to the original object file and stop processing.
  - Start a distiller process for the canonical form of the object.
- If a distiller process has finished, fetch the resulting object.
  - In case it has any contents, copy the object to the object pool and store its ID to both MD5 and URL databases.
  - If there are any references to URL's, consult the URL database and if they are unknown, add them there and to the URL queue, too, possibly creating new host objects.
  - If there are any references to new objects, add them to the object queue.

4.3. Queue Management

The queue is stored in two files: `db/hosts` and `db/queue`. The `hosts` file contains a list of all known hosts (in reality host-port-protocol triplets) together with their host ID and position of the first queued record in the `queue` file. The `queue` file is divided to one-kilobyte pages, each one containing a sequence of queued paths accompanied by a pointer to a next page for the same host. There is also a dummy host with an empty name holding all unused file pages.

4.4. Downloaders

Each downloader process receives from one to three arguments: the URL to be downloaded, an optional date of last modification of the object (the document has to be retrieved again only if it's newer than this date) and an optional old version fallback flag (set if an older version of the protocol shall be used).

The downloader should send the object description to its standard output and all error messages to the standard error stream. Standard input should not be used.

This version offers downloaders for HTTP 1.1 and local files (the `file` URL scheme).

4.5. Data Type Handling

If the server has supplied any `Content-Type` and `Content-Encoding` header fields, we attempt to honor them and extract from them as much information as we can. We work with three basic characteristics of the document here: content data type, encoding and character set. The character set handling is described in a separate section below, the rest follows (all data needed by this process are present in a configuration file):

- We perform type translation first. Several types are known to be unreliable or to be only aliases for other types. We have a user-definable table of type exceptions handling these problems.
- In case we have received no data type information, we start guessing. Not a usual crystal ball vaticination, but simple pattern matching on both file name and file contents (only first 256 bytes in order to simplify and accelerate the process).

- We can map some data types to content encodings. For example, `application/gzip` gets translated to an unknown data type encoded by `gzip` encoding.
- Several encodings are mere aliases (`x-gzip` is equivalent to `gzip`). We canonify them first.
- If the file has a non-null content encoding, we decode it and restart the data type processing with a modified file name (encoding-specific extensions (e.g., '`.gz`') are stipped off). If the encoding is unknown, we throw the object out and mark it as invalid as we don't know how to handle it.
- In case the primary data type is `text/html`, we search for embedded `<META>` tags with the `HTTP-EQUIV` attribute set to `Content-Type` and extract new content type and charset info from them. If we have succeeded, we restart the data type processing, skipping this part in the next pass (loop avoidance). This is a very stupid 'extension' to the HTML standard and its author should be hanged immediately on the nearest gallows, but unfortunately, lots of people are using it nowadays...
- If the data type is known to require charset info, we start the character set selection process.

4.6. Character Set Handling

Current World Wide Web is full of strange and mostly faulty character set selection and recoding mechanisms (see `SaCzech` for a good example of what to avoid), so the usual charset negotation techniques described in the `HTTP` standard are not sufficient for us. We try to use the following algorithm and only the experience will tell us whether it's good or not – it's an ugly heuristics, but it should work in most cases:

- If the server has given us no language list, we try to guess it from a domain name (typically `en` plus a local language of the domain).
- Then we select character sets known as being used for the specified languages together with the character set reported by the server (which unfortunately cannot be blindly followed as there are lots of faulty and/or misconfigured servers).
- For each language, we have a set of typical and forbidden characters. We merge these sets for all defined languages.
- We start scanning the file in 1KB blocks. In each block we try to convert character codes according to all selected character sets and count typical and forbidden characters for each of them.
- If the server has reported any character set, we look at the counters for this set and try to decide whether the document seems to be in that charset or not. If not, we ignore the supplied information and continue with the selection process. If yes, we accept this charset and exit the process.
- If any charset is known to be invalid (due to a non-zero number of forbidden codes), we remove it from the list. If the list becomes empty, we exit the selection with a fallback character set (the charset supplied by the server if there's any or a language-dependent default otherwise).
- If any charset shows significantly more typical characters than the others (what does 'significantly' mean is a matter of heuristics and should be tuned manually to obtain a reasonable tradeoff between performance and reliability), we exit the selection with that charset as the result.
- Continue with the next file block until one of the rules exits the process or the file ends.
- If the file has ended, we take the *first* one of the charsets remaining selected valid for the *first* language specified on the list (allowing the user to define preferred charsets by placing them *before* the others).

4.7. Distilling

Each distiller gets two arguments: the URL of the document and the base URL of the document (can be got from document headers by the downloader, otherwise it's identical to the document URL).

It reads the data file from its standard input and writes new object data to standard output and log messages to standard error stream.

This version supports analysis of HTML (content type '`text/html`') and plain text files ('`text/plain`').

4.8. URL Filters

The gatherer daemon offers a simple URL filtering mechanism. The filter is a sequence of `Allow` and `Deny` rules, each followed by patterns for protocol name, host name, port number and relative path. The first matching rule is used. The filter should end with a pattern matching all remaining URL's and specifying whether to permit or forbid them.

4.9. Object Expiry and Refresh

The algorithms mentioned above are good for one-shot gathering, but not for continuous refreshing and updating of the database. The following extensions are implemented in order to accomplish desirable behaviour in such cases:

* A refresh timeout is defined. If we get a reference to an already known object and it has been downloaded before more that this timeout, we re-queue it again in order to verify it.
* If we have to download an already known object, we use a `If-Modified-Since` request in case the protocol supports it. If it says the object was not modified, we only update its download time. If it has changed, we remove the old object and replace it by the new contents. And if it has ceased to exist, we only remove the old object and modify the object database to hold the unreachability information instead. Warning: if a document is available under more different URL's, we must process each URL separately. In such cases, we cannot remove the object itself, but only one URL reference in it and delay object removal until the last URL disappears (we sometimes re-queue the other URL's as they probably have changed, too).
* An `expire` utility is ran periodically. It scans the URL database, removes all URL's older than an expiry timeout (dependent on whether it's a normal document or an error marker) and re-queues all non-expired URL's older than a retry timeout. The multi-URL problems are solved in the same manner.

4.10. The Object Transport Daemon

In addition to the gatherer daemon, there is also an object transport daemon (commonly known as '`gived`'), which is responsible for handing all known objects out to one or more search engines. It accepts connections via the `TCP`, gets start times from them and sends all objects newer than the start time to the connection. Unfinished objects are excluded from this process (they are temporarily marked by the gatherer daemon as unreadable for the others).

5. The Search Engine

The search engine maintains its own indexed database of objects obtained from several gatherers either locally or via the network and offers document search services. The experience has shown that dynamic updating of this database is not desirable (see also the Harvest project) as it decreases data structure efficiency, so we always build the whole database from scratch.

The search engine can be clearly divided to three parts:

- The database builder (`dbuild` et al.).
- The search daemon answering to generic search queries coming from various sources via the TCP (`sherlockd`).
- The WWW interface to the search daemon (`scgi`).

5.1. Database Structure

The search engine indexed database consists of the following parts:

- Object pool file – this pool holds detailed information on all objects known to the search engine. It's created in the extraction phase and contains selected parts of all objects offered by the gatherer's object transport daemon. The objects are stored one after another, each one terminated by a null byte and numbered sequentially starting with zero (as it's a case of re-indexed objects, we follow the `O` attribute semantics described in the Object Interchange Format chapter).
- Object pool index – a straight array of object start offsets in the object pool file followed by length of that file. This allows objects known by their new (re-indexed) numbers to be extracted from the pool quickly.
- Reference list – for each known word or non-word (non-words are special entities such as URL's which are indexed in whole and not split to words), we have a sequence of references sorted by object ID's here (the first long word contains number of references). Each reference has the following format (all multi-byte quantities are stored in the machine's native endianity, because these files are not needed to be portable):
    * New (re-indexed) ID of the object we refer to (32-bit integer).

    * Section tag (each data source can be tagged in order to allow the search to be restricted to specific sources) (8-bit integer).
    * Quality factor (5-bit integer: 0=worst, ..., 31=best) – see description of the `W` object attribute.
    * Reference type (3-bit integer: 0=word, others are various kinds of non-word objects (e.g., URL references)).
- Word list – list of all known words together with pointers to their blocks in the reference list file. The words are divided to groups having the same signature (the signature consists of uppercased unaccented variants of first 8 characters), each word being described by the following parameters:
    - Word name (a null-terminated sequence of uppercase characters), empty for end of group.
    - File pointer to the reference list (unaligned 32-bit integer).
- Word index – a B-tree containing all known word signatures together with their word list group pointers. As the B-tree is built statically, its shape can be controlled by a specific node allocation strategy to avoid unbalanced subtrees and to allow omission of all node pointers from the file (all of them can be deduced back if the reader knows the allocation strategy used by the builder). For simple searches, it would be better to use a statical hashing as used for the non-words, but this structure allows suffix matches to be optimized (we can look up both previous and next signature in the B-tree as opposed to the hash table).
- Non-word list – list of all non-word keys (those having non-zero reference type) grouped by their hash value and stored in the same format as the word list.
- Non-word index – array of pointers to non-word groups indexed by their hash values.

5.2. Database Builder

Building of the search database is done in several consecutive steps controlled by the `dbuild` script:

- `objget` – Getting of objects from the object transfer daemons of all known gatherers. The objects are analysed, re-indexed (new object numbers starting with 0 are introduced), unnecessary attributes are removed and everything else is split to four files: the object pool, the object pool index (both in the final form as described above), references to words and references to non-words (both of these lists contain all known (non-)words accompanied by a list of references to them [the format of references is the same as for the final reference list file], but each entry can occur multiple times [only local optimization and pre-sorting are done in this step]).
- `sword` – Sorting of both reference files. Straight 2-to-2 file mergesort with merging of references to the same (non-)word whereever possible. After this pass, each (non-)word is unique. Words are sorted by their signatures and within each signature group lexicographically (needed for the B-tree and signature list concept), non-words are sorted lexicographically (it would be better to sort them by their hash values to simplify building of the hash table, but we unfortunatelly cannot as we don't know the number of non-words which affects the hash table size).
- `splitter` – Takes both word references and non-word references and creates the final reference list, the final word list, a signature list and an intermediate non-word list.
- `wtree` – Creates the signature B-tree from the signature list.
- `nhash` – Constructs the non-word list and non-word index. Now we have enough information about the non-words, so we calculate the appropriate hash table size and use mergesort to sort the non-words by their hash value.

5.3. The Search Daemon

The search daemon maintains a queue of search requests coming from different sources across the network or from the local host and attempts to serve them in a FIFO order.

Each query is represented in a very simple query language:

* *query* ↦ *[section] [from] [to] expr*
* *section* ↦ `SECTION` *comma-separated-list-of-numbers*
* *expr* ↦ *word* | *reference* | `(` *expr* `)` | *expr* `AND` *expr* | *expr* `OR` *expr* | `NOT` *expr*
* *word* ↦ *[quality]* `"`*pattern*`"`
* *reference* ↦ *[quality] ref-keyword* `"`*url*`"`

* $pattern \mapsto patchar \mid pattern\ patchar$
* $patchar \mapsto char \mid$ `*` $\mid$ `?`
* $string \mapsto char \mid string\ char$
* $quality \mapsto number$
* $from \mapsto$ `FROM` $number$
* $to \mapsto$ `TO` $number$

where *section* limits documents by section number, *to* specifies a limit on size of resulting document set, *from* specifies which document on the resulting list should be reported as the first one (i.e., how many documents from the start should be ignored), *category* limits reference category for non-words.

Processing of each query is very simple: we split the query to a word pattern list, a non-word pattern list and a boolean expression between pattern occurences. Then we scan the indices, the lists and the reference list and find all documents containing at least one of the words and matching the category and section requirements (it's simple as all references are sorted by object ID's). For each document found, we evaluate the boolean expression and if it's true, we calculate a document quality factor from the quality factors known for the references and given in the pattern and funnel them through a heap, selecting the best *to* documents.

5.4. The WWW Interface

The WWW Interface is a simple `CGI` script serving as an interface between the World Wide Web and the search daemon. It translates the query form posts to search requests and the search results to `HTML`.

6. References

`http://info.webcrawler.com/mak/projects/robots/guidelines.html` – Guidelines for WWW Robot Writers, the Robot Exclusion standard and references to other information sources.
`http://harvest.cs.colorado.edu/` – The Harvest project
`http://glimpse.cs.arizona.edu/` – Glimpse (Global Implicit Search)
`http://www.unicode.org/` – The UNICODE standard
RFC 2068: The Hypertext Transfer Protocol, version 1.1
RFC 1866: The Hypertext Markup Language, version 2.0
RFC 1738: Uniform Resource Locators
RFC 1808: Relative Uniform Resource Locators
RFC 1402: There's Gold In Them Thar Networks! or Searching for Treasure in all the Wrong Places
RFC 1321: The MD5 Message-Digest Algorithm
RFC 2044: UTF-8, a transportation format of Unicode and ISO 10646
RFC 1591: Domain Name System Structure and Delegation
RFC 1950: ZLIB Compressed Data Format Specification version 3.3
RFC 1951: DEFLATE Compressed Data Format Specification version 1.3
RFC 1952: GZIP file format specification version 4.3
RFC 959: File Transfer Protocol (FTP)
RFC 2045 – 2049: Multipurpose Internet Mail Extensions