Customer: Kjell Lemström

Instructor: Anita Kasari

OT4: pics99

Implementation and Testing Document

Timo Immonen
Jussi Lindgren
Sampo Pyysalo

Helsinki May 27, 1999

Implementation and Testing Document
Software Engineering Project
UNIVERSITY OF HELSINKI
Department of Computer Science

Contents

Version Number	Comments	Date/Author
0.1	document started	27.4.99 /TI
0.2	gui text added	29.4.99 /TI
0.3	alglib, misc, libtest, perf added	29.4.99 /JL
0.4	gui testing added (partial)	10.5.99 /TI
0.5	gui testing done	14.5.99 /TI
0.6	added maint, changes, misc	23.5.99 /JL
0.7	added db and xdr and tests	27.5.99 /SP

1 Introduction (TI)

PICSearch is a development tool for content-based image retrieval algorithms. It is aimed to serve as a testing platform for new retrieval algorithms. It can also be used as a conventional image cataloging program.

The initial version PICSearch was developed at Helsinki University of Technology. First version, **PICSearch v1.0**[1], was restricted to use a local image database only. Further development at University of Helsinki resulted in **PICSearch v2.0**[2], a version enhanced with rudimentary network capabilities. A third version of program was also developed at University of Helsinki. **PICSearch v3.0**[3] added a cache implementation for greater network efficiency and also featured an improved graphical user interface.

This document reviews implementation work and testing done during the second phase of the pics99 project. New features introduced are an independent search algorithm entry method and a more standard database search engine. Internal communication has also been clarified. User interface has also been updated with necessary options for algorithm entry method.

In section 2 changes to the Design Specification are curtly discussed. Section 3 describes the new database implementation, and configuration files. In section 4 the new internal communication is discussed. Section 5 contains descriptions of the new graphical user interface (GUI) elements. New, independently created search algorithm libraries supported by **PICSearch v3.1** are described in section 6. Miscellaneous changes to the system are listed in section 7. Sections 8, 9, 10 and 11 contain information of testing performed for GUI, algorithm libraries, internal communication and database subsystem, respectively. Section 12 gives an assessment of the system performance, and finally section 13 contains useful maintenance advice for persons undertaking further development of PICSearch.

2 Changes to the Design Document (JL)

Some things specified in the previous design document were found to be inadequate or unsuitable during the development phase.

- Tk parameter window code

  There was no good reason to store the algorithm Tk parameter window code to the database. Instead, its kept in the database/ directory along with the respective algorithm library, having a name "alg[ID].tk". The code can be easily read to memory when needed, by just knowing the algorithm ID.

- Saving of various picture formats

  It wasn't specified in the previous documents, but **PICSearch v3.1** will no longer work as a picture conversion tool. Instead, PICSearch tries to preserve the data in its original format. Thus, a picture inserted to the database as JPEG will emerge as identitical JPEG file on save. This applies to the other formats supported by PICSearch as well.

  Thumbnails will be saved as JPEGs.

  Those who need to convert pictures should use some external utility package best suited for that purpose. Both GUI and command-line based tools are freely available from the internet.

- ID and Name search algorithms

  The ID and Name algorithms performed rather badly in v3.0 and although they we converted to the new library format, their operation was never fixed.

3 Database (SP)

The PICSearch database implementation has three separate parts: A simple configuration file-based system to handle user, algorithm and image database management, a GDBM-based system to handle the signatures and thumbnails of each separate image database and a separate top-level class that brings together their functionality under a relatively simple interface. This section will describe each of these parts in turn.

3.1 The Configuration Files

A PICSearch configuration file is divided into sections which contain a number of named variables which hold string values. The basic syntax of each configuration file is designed to be simple enough to be understood, modified and written by human, script, and PICSearch alike. The sections are represented by bracketed strings, variables and their values are represented by two strings on a single line separated by the first '=' sign on that line, for example:

```
[section1]
variable1=value1
variable2=value2
```

```
[section2]
variable1=value3
```

As indicated by the example, variables in separate sections may have the same name and hold separate values.

3.1.1 Configuration File Structure

PICSearch has four different configuration files. One describes the users of the system and their respective rights, another lists the separate image databases in the system, a third one contains the algorithms that are present and in use by the system and the very last one has some general parameters for the server.

The structure of each of these configuration will now be detailed below, for your extreme discomfort.

- users.cfg: each section describes a single user and has the same name as that user. Each section holds the following information:
  - authLevel: the authorization level of this user to the server. Legal values are 0, 1 and two, which represent no access, normal user access and superuser access, respectively.
  - password: the user's password.
  - changePass: indicates whether or not the user is allowed to change her password. Values 1 and 0 represent yes and no, respectively.
  - DB-*DBID*: the authorization level of this user to the database that has ID *DBID*. There are as many of these fields as there are databases. Values 0,1,2 and 3 represent no access, thumbnail view access, full read access and full write access, respectively.
- DBs.cfg: each section describes a single image database and has the same name as that image database. Each section holds the following information:
  - directory: the directory on the server's filesystem that the database resides in.
  - ID: identifier number of the database. Integer values greater than 0 are allowed.
  - alg-*algID*: indicates whether or not the algorithm with ID *algID* is available in this database. Values 1 and zero mean exactly what you'd suppose they would.

- algs.cfg: each section describes a single algorithm and has the same name as that algorithm's ID. Each section holds the following information:
  - algClass: the class of the algorithm. Possible values range 0 through 6 and represent a number of alternatives for the algorithm class that I'm too lazy to look up right now.
  - paramWin: indicated whether or not there is a separate tcl/tk parameter window for this algorithm available. 1 or 0.
  - active: indicates whether or not the algorithm is currently active. 1 or 0.
  - replacedWithID: if an algorithm has been replaced by a newer one, this field contains that algorithm's ID, 0 otherwise.
- params.cfg: currently defines only a single section called host, which contains the following host configuration parameters:
  - http: URL to someplace containing information about this server. Or then again, maybe not.
  - info: string printed on client startup.
  - maxclients: maximum number of clients allowed on this server simultaneously.
  - timeout: time, in minutes, that server waits for client messages (if none are arriving) before assuming the client dead and disconnecting.

3.1.2 ConfigFile Implementation

A separate configFile class was introduced to handle basic configuration file operations. The basic interface supports creation, reading and saving of configuration files as well as modifying the sections and variables stored in them.

The internal representation is a two-level C++ standard template library (STL) map, mapping first section names into sections and then variable names into their values. As all of these are represented internally by strings, this two-level map can be used as shown below:

```
cfgFile[``sectionName''][``variableName''] = ``value'';
```

Which would effectively assign the value "value" to the variable named "variableName" in section "sectionName". This simple design is complicated slightly by the need to write all modifications into the actual configuration file, but the problem for semi-static configuration files is easy to solve: The configuration files are updated only on separate request from the server (not each time some other process makes a modification) and mutual exclusion for writes is handled by making all update operations atomic on the filesystem level.

A close approximation of the configFile class interface is shown below:

```
class configFile {
  //..private/protected members
public:
  configFile(string fName);

  void save(void);
  void load(void);
  void clear(void);
  void reload(void);

  cfgIterator begin(void);
  cfgIterator end(void);
  cfgIterator find(string k);
  bool exists(string k);
  size_t size(void);

  int remove(string sec);
  int remove(string sec,string var);
  void set(string sec,string var,string val);
  string get(string sec,string var) const;
}
```

The operations are rather self-explanatory, but a brief summary of each is presented below just to bore you:

- configFile(fName): the constructor takes the file name as its parameter, creates the file if it doesn't exist and reads in its contents if it does.
- save(void): Saves the configFile. As this overwrites the current file, it should only be done when no other processes have modified the configFile since it was last read.
- clear(void): Clears all entries in the configFile.

- reload(void): Clears all configFile entries and fills them in from the file specified on creating the configFile.

The following few operations relate to iterating through the configFile, which means looking at each of the sections in sequential order. The cfgIterator is a simple little class that implements a pointer-like interface and will *not* be discussed here.

- begin(void) returns an iterator "pointing" to the first section in the configFile.
- end(void) returns an iterator "pointing" just beyond the last section in the configFile.
- find(key) returns an iterator "pointing" to the section with the name given as parameter, if it exists.
- exists(key) returns true if a section with the name given as parameter exists, false otherwise.
- size(void) returns the number of sections in the configFile.

The following operations allow removing sections and variables as well as reading and modifying the values of variables.

- remove(sec) removes the section specified.
- remove(sec,var) removes the variable specified.
- set(sec,var,val) sets the value of a variable.
- get(sec,var) return the value of a variable.

In addition to this generic configFile class (which provides all functionality needed to handle the configuration files) a number of specialized classes are provided explicitly for the purpose of restricting the interface to only allow storing variables with specific names (for example "ID") or specific values (for example only strings that represent integers for the variable "ID"). These will not be described here.

3.2 The Image Databases

A single PICServer may provide access to a number of separate image databases. Each of these holds one file containing thumbnails of images in the database and a file for each algorithm's signatures. There is also a separate directory containing all the images as separate files in their original format.

The file that contains the thumbnails and the files that contain the signatures are created and managed using the GNU database management (GDBM) system, which supports variable-length keys and data as well as a simple but sufficient form of multiple-readers, single-writer locking.

3.2.1 Image Database Structure

As mentioned above, there are two different types of file (or, two different types of database table) in an image database: One that holds thumbnails another that holds signatures. Both types of table have a single key value, the ID number of a picture.

The first type of table (of which there is exactly one in each image database) essentially contains each image's thumbnail and some additional information. The whole structure is called "picture" and is presented below:

```
struct picture {
  char fileName[PATH_MAX+1];
  char paramStr1[PARAMSTRING_MAX+1];
  char paramStr2[PARAMSTRING_MAX+1];
  unsigned int thumbSize;
  char thumbNail[0];
};
```

For an explanation of the array with size 0, see below. The fields' meanings are as follows:

- fileName: a static array large enough to hold the longest pathname that the local system can accommodate and a null terminator. Holds the name of the file that this picture is stored in.
- paramStr1 (and 2): static arrays large enough to hold the largest parameter string allowed on the compilation of PICSearch and a null terminator (PARAMSTRING_MAX is a compile-time constant). Meant to hold auxiliary information on the picture, currently not in use.
- thumbSize: the size of the thumbnail of the picture in bytes.

- thumbNail: a variable-length array that holds a thumbnail of the image.

The second type of table (of which there is one for each algorithm in the image database) simply contains a signature for each picture in the image database:

```
struct signature {
  unsigned int sigSize;
  char sig[0];
};
```

For an explanation of the array with size 0, see below. The fields' meanings are as follows:

- sigSize: the size of the signature in bytes
- sig: a variable-length array that holds the signature data.

3.2.2 Image Database Implementation

The image database defines a dbHandler class that provides all the functionality needed to create and maintain the database tables described above. Each dbHandler object handles a single GDBM file (and therefore a single table) so one is needed for the thumbnail table and for each of the signature tables.

The dbHandler class itself is implemented in a very straightforward fashion by delegating all operations to GDBM. The heavy reliance on GDBM can be seen in the dbHandler interface, which defines all operations in terms of *datums*, the basic GDBM structure for keys and data. A datum has the following simple structure:

```
struct datum {
  char *dptr;
  int   dsize;
}
```

The structure fields have the obvious meaning:

- dptr (for *data pointer*): a pointer to the data associated with this datum.
- dsize (for *data size*): the size of the data pointed to by dptr, in bytes.

Using datums (and, specifically, their dptr fields) to refer to the picture and signature structures above requires heavy use of casting and some creative memory management. These implementation details are hidden from the end user by the main database class interface (see section FOO). GDBM's reliance on datums also explains the odd way the picture and signature structures are defined (see above). The signature structure is shown again below:

```
struct signature {
  unsigned int sigSize;
  char sig[0];
};
```

Since GDBM only accepts data that is stored as a single continuous block of memory it is necessary to have the final structures handed to GDBM be represented as such. To avoid copying, those same structures are also handed to the end user as fetch operation return values, a minor inconvenience. Having an array of size 0 as the last data member of a structure is a common C idiom used to create a data member of varying size. The strategy works because of the ANSI C (and C++) guarantee that the order of structure members is not altered. It does complicate memory management slightly and is easy to get wrong, but again the main database class can be used to hide these details from the end user.

Having thus introduced datums and some GDBM design considerations, the dbHandler class will now be presented in relevant detail:

```
class dbHandler {
  //..private/protected members
public:
  dbHandler(string fName);
  ~dbHandler(void);

  datum fetch(datum k);
  bool store(datum k, datum d);
  bool remove(datum k);
  datum firstKey(void);
  datum nextKey(datum k);
};
```

All of the methods (except for the constructor and destructor) are implemented simply by delegating the task to GDBM. Refer to GDBM documentation for details, a prief overview is presenter below:

- dbHandler(fName): the constructor takes a single parameter, the name of the file to access.
- dbHandler(void): the destructor performs any and all necessary cleanup.
- fetch(key): return a datum containing the data of the item referenced by the key contained in the datum given as parameter.
- store(key, data): store the data contained in the datum d under the key contained in datum k.
- remove(key): remove the item referenced by the key contained in the datum given as parameter.
- firstKey(void): return a datum containing value of the key referencing the first item in the database.
- nextKey(key): return a datum containing the value of the key referencing the item that follows (in the database) the item referenced by the datum given as parameter.

The firstKey and nextKey functions are used for sequential access to the database when going through all the items stored in the database.

3.2.3 Image Database Interface

The dbHandler class provides all the relevant funtionality needed to store, modify and fetch thumbnails and signatures, but the interface is clumsy and having the end user deal direcly with datums introduces some unpleasant implementation details and exposes her to memory management issues. PICSearch, in its never-ending quest to utterly and completely satisfy the user, has a separate class to handle these unfortunate details of interfacing with the dbHandler. This class is called imageDB, and it provides some extra functionality just for fun (and profit!).

The imageDB interface with the boring bits cut out:

```
class imageDB {
  //..private/protected members
public:
  bool addSigDB(algorithmID_t alg);
```

```
  bool removeSigDB(algorithmID_t alg);

  bool  saveImage(string fileName,image img);
  image loadImage(string fileName);

  picture *fetchPic(pictureID_t ID);
  bool storePic(pictureID_t ID,const char *fileName, const char *paramStr1,
                const char *paramStr2,unsigned int thumbSize,
                const char *thumbNail);
  bool removePic(pictureID_t ID);
  pictureID_t firstPicKey(void);
  pictureID_t nextPicKey(pictureID_t oldKey);

  signature *fetchSig(algorithmID_t alg,pictureID_t pic);
  bool storeSig(algorithmID_t alg,pictureID_t pic,
                unsigned int sigSize,const char *sig);
  bool removeSig(algorithmID_t alg,pictureID_t pic);
  pictureID_t firstSigKey(algorithmID_t alg);
  pictureID_t nextSigKey(algorithmID_t alg,pictureID_t oldKey);
};
```

The largish interface can make the class seem complicated, but the individual operations are simple:

- saveImage(fileName,image): saves the image given as parameter under the file name given as parameter. *image* is a very simple structure with an unsigned integer size field and a pointer to character field to address the image data.
- loadImage(fileName): loads in image from the file with the name given as parameter.

As mentioned before, PICSearch stores pictures separately in their original format. The above two funtions perform the simple task of transferring a picture between disk and memory using standard file I/O.

- addSigDB(alg): adds a signature database with the ID given as parameter.
- removeSigDB(alg): removes the signature database with the ID given as parameter.

- fetchPic, storePic, removePic, firstPicKey and nextPicKey: these functions simply present a more pleasant interface to the user, delegating all their operations straight to the dbHandler in charge of storing the picture items (referred to earlier as just thumbnails).
- fetchSig, storeSig, removeSig, firstSigKey, nextSigKey: the same applies to these functions with the small difference that they address a specific signature database (given as parameter) rather than the single common picture database of this imageDB.

These operations need not be examined in further detail since they have essentially been explained above along with dbHandler.

3.3 Interface to the Database System

The configuration files and the image databases provide all the operations needed for the PICSearch database implementation. What they are lacking is centralized point of access and control, which is provided by a separate class called newPICDB. Since the image database interface is full and safe as it stands and the configuration files come supplied with wrappers to make them simple to use correctly and difficult to abuse, The newPICDB class can provide almost direct access to objects of these classes, leaving mainly the task of managing their creation, removal and maintainance. A separate issue That had to be addressed in the newPICDB was finding new ID numbers for databases and algorithms; since DB and algorithm information is stored in configuration files which do not provide read-locking the configFiles themselves are vulnerable to race conditions (two separate servers adding a database at the same time might choose the same ID number), and the creation of unique IDs is therefore a part of the newPICDB services.

The class interface in relevant detail follows:

```
class newPICDB {
  //..private/protected members
public:
  DBConfigFile DBs;
  userConfigFile users;
  algoConfigFile algs;
  paramConfigFile params;

  bool addImgDB(databaseID_t DBID,string dir);
```

```
  bool removeImgDB(databaseID_t ID);
  imageDB *getImgDB(databaseID_t ID);

  algorithmID_t newAlgID(void);
  databaseID_t newDBID(void);
  pictureID_t newPicID(databaseID_t DB);

  newPICDB(void);
};
```

As can be clearly seen from the interface, specializations of the configFile class can be accessed directly, and there is a function that returns pointers to image databases. These classes can be used "as is" without need for intervention from the newPICDB class (as explained above) with the exception of managing image databases, which can only be added and removed through separate functions. The static four configuration files need no such maintainance. A brief and annoying as well as delightfully space-consuming explanation of each of the members follows:

- DBConfigFile DBs: a configuration file restricted to dealing with stored database information as explained above in the section that details the various types of configFiles.
- userConfigFile users: a configuration file restricted to dealing with user information.
- algoConfigFile algs: a configuration file restricted to dealing with algorithm information.
- paramConfigFile params: a configuration file restricted to dealing with server parameters.

It needs to be noted that since the configFile base class can contain any data presentable in string form, all of the specialized configFile types are restrictions of, not extensions of, its functionality which makes them uninteresting for this examination.

- addImgDB(DBID, dir): adds a new image db with ID DBID in the directory dir, unless either one is already in use.
- removeImgDB(ID): removes the image db with the ID given as parameter.
- getImgDB(ID): return the address of the image database object with the ID given as parameter, or 0 if it doesn't exist.

These functions preserve control over how image databases are created, removed and stored inside the newPICDB class while giving users full access to the image databases in the system.

- newAlgID(void): returns a unique algorithm identifier.
- newDBID(void): returns a unique database identifier
- newPicID(DB): returns a unique picture identifier in the context of the database specified.

The newPICDB class is able to provide this functionality in a controlled fashion by using GDBM's locking capabilities.

4 Client-Server Communication (SP)

The PICSearch 3.1 project featured an extensive rewrite of the program's communication subsystem. As before, all communication between the PICSearch client and server is performed over TCP/IP, which provides a reliable byte stream to the system. The previous PICSearch communication system augmented this service by providing a messaging service that incorporated message typing (though in a weak way) and a platform-independent way of sending basic data types that was essentially limited to sending integers of 8, 16 or 32 bytes with a known byte order.

4.1 Communication Structure

The weakness of the type system and a problem with the messages' fundamental lack of structure prompted a move to use external data representation (XDR) for client-server communication. XDR provides facilities to automatically generate functions to write arbitrary structures to and read them from a byte stream. In addition to creating these functions to serialize and deserialize structures, the XDR package provides a number of calls that can be used to simulate a record stream which makes it possible, among other things, to skip an unexpected, unknown or faulty record on the stream.

Merely using XDR would have left open problems with typing (XDR provides none) and would have unnecessarily burdened the end user with the need to directly call the admittedly clumsy XDR routines. An extra layer was therefore added to insulate the end user from the inner workings of the system. This layer wraps the XDR routines and provides message typing in addition to a bit of syntactic sugar. The extra layer is represented in the program by a class called xdrHandler.

4.1.1 Message Layer Services

The xdrHandler class provides C++ wrappers for all the xdr structures in the program plus message typing that can be used to identify the separate messages when receiving them. The basic operations provided by xdrHandler are sending a message, receiving a message, and freeing the memory reserved for a message. A separate operation for freeing memory is needed because the messages are potentially large and their size may not be known in advance, so memory reservation is handled by the XDR routines. This means that the user has to make a separate call to free that memory once she is done with it, which is not terribly difficult to do once you get used to it.

To make these three operations simple to use a decision was made to overload the send, receive and free functions separately for each structure that may be sent over the network. These functions are all similar in structure and rather simple. The form of each will be presented using the acknowledgment message as an example. The send functions take the following form:

```
bool send(ack_msg *msg) {
  MessageType type=msg_ack;

  xdrHandle.x_op=XDR_ENCODE;
  return(xdr_MessageType(&xdrHandle,&type) &&
         xdr_ack_msg(&xdrHandle,msg) &&
         xdrrec_endofrecord(&xdrHandle,true));
}
```

The function takes a pointer to the message to be sent as a parameter. A MessageType variable (defined by xdrHandler) is set to a value identifying the message type (an enumeration, also defined by xdrHandler), ack in this case. A field indicating the operation type in an XDR handle type is then set to a value indicating that the function wishes to perform an encoding operation (all of these values are defined by XDR). The final statement calls xdr-MessageType (generated by XDR) to encode the message type, if this call succeeds it proceeds to call the XDR-generated function to encode the actural message being sent and if this also succeeds it finishes of by calling xdrrec-endofrecord (defined by XDR), which marks the end of a record and optionally places it on the underlying stream depending on its last parameter (passing true as shown here sends the message straight away).

The receive functions have the following general structure:

```
bool xdrHandler::receive(ack_msg *msg) {
```

```
  xdrHandle.x_op=XDR_DECODE;
  if(currentMessage()!=msg_ack)
    return false;
  proceed();
  return(xdr_ack_msg(&xdrHandle,msg));
}
```

The operation field is first set to indicate a decode operation, after which currentMessage (defined by xdrHandler) is called and its return value (which indicates the type of the current message) is compared against the type of the message which the function attempts to receive. If they match, proceed (defined by xdrHandler) is called so the next call to a receive function will fetch the next message regardless of the success of this operation. Finally, the XDR-generated function is called to decode the message and its return value is passed back to the user.

The free-functions are the simplest of the three:

```
void xdrHandler::free(ack_msg *msg) {
  xdr_free((xdrproc_t)xdr_ack_msg,(char *)msg);
}
```

They simply delegate the task of freeing the memory to a corresponding XDR-supplied routine.

4.1.2 Message Layer Implementation

As PICSearch currently defines more than 20 different messages, writing separate send, receive and free routines for each would be tedious, error-prone and just plain stupid. Since the routines all take a standard form and can be easily generated, the XDR approach was followed and a separate program, *msggen*, was written to generate these routines from a simple language defining the messages (actually just a file that lists their names). The msggen "utility" is included, with full source, in the picsearch distribution and is incorporated in the picsearch make system in a way that makes its presence transparent to the end user. All that is needed to add a message to the PICSearch system is to define it (using the XDR data definition language) in the file that holds the PICSearch message definitions (message.x) and include its name in a separate file (message.lst) that msggen uses for input.

In addition to the generated send, receive and free routines, xdrHandler provides the following easy-to-use interface (irrelevant detail suppressed):

```
class xdrHandler {
  //..private/protected members
public:
  xdrHandler(int connfd);
  ~xdrHandler(void);
  MessageType currentMessage(void);
  void proceed(void);
  //..send, receive & free functions
}
```

The services are simple and few:

- xdrHandler(connfd): establishes an xdr message stream on top of a previously opened byte stream identified by the file descriptor number given as parameter.
- xdrHandler(void): closes the message stream, but not the byte stream which it was based on.
- currentMessage(void): returns a value identifying the next message that can be received from the message stream. Can be used to construct a massive and unwieldy switch to handle all possible messages.
- proceed(void): skips the current message and proceeds to the next one in the message stream. Can be used to skip unknown or unexpected messages.

As indicated above, xdrHandler doesn't create sockets or manage communications itself, but attaches on top of an existing connection instead. This design allows the typed message service to be independent of the underlying data stream and promotes flexibility and design reuse (nah, just kidding).

4.2 Communication Update

Changing all communications from the old system to the new touched a lot of files and functions, and since the server-side operations that perform communication are also directly responsible for database access, many of the functions were also modified during the database system rewrite, effectively causing a full rewrite of those functions, and, at times, whole files. When appropriate, separate modifications to functions will not be listed but rather stated as "function rewritten" or even "file rewritten".

4.2.1 Server Modifications

Files that were touched under the server directory:

- inc/PICMessage.h: the file was removed completely. Its resident class, the 442-line PICMessage, was seriously cramping the style of most functions that used it.
- inc/PICServer.h: A number of methods in the PICServer class were modified. The modifications include the following:
  - TranslateMessage was removed and replaced by xdrTranslate.
  - AddDB and RemoveDB were removed as unused and unnecessary.
  - Query, StopQuery, SendMoreNails, InsertPicture, SendPicture, DeletePicture, AdjustAlgorithms, CreateDB, DeleteDB, ChangeUserRights, AddUser, RemoveUser, ChangePassword and SetOptions were modified to take corresponding message parameter instead of the generic single PICMessage object parameter they used to take.
  - SendCrippledInit was removed as unnecessary.
- src/PICServer.cpp: was renamed to Server.cpp and the following things were changed:
  - Main was modified to conform to the new communication system.
  - InitOptions, Initialize, SendFail and SendComplete were rewritten to use the new communication system and database.
- src/Database.cpp: the file went through a complete rewrite. All methods in the file were chaged, in the following ways:
  - AddDB and RemoveDB were removed as unused and unnecessary.
  - AdjustAlgorithms, CreateDB and DeleteDB were rewritten to use the new communication system and database.
- src/Translate.cpp: the file was completely rewritten, the only function in had in it, Translate, was removed and replaced by xdrTranslate, which does the same sort of job.
- src/Query.cpp: the following functions were modified:
  - Query was rewritten to use new comm and DB.
  - SendQueryProgress was rewritten to use new comm.
  - SendNails was rewritten to use new comm and DB.

  - SendMoreNails was rewritten to use new comm.

- src/Options.cpp: the file was completely rewritten to utilize the new systems, which both its functions, SendOptions and SetOptions, use heavily.

- src/User.cpp: the file was completely rewritten. The following functions are still included in their new form: SendUsers, ChangeUserRights, AddUser, RemoveUser and ChangePassword.

- src/Picture.cpp: the following functions were rewritten:

  - InsertPicture: new comm and DB.
  - SendPicture: new comm and DB.
  - DeletePicture: new comm and DB.

- src/Maint.cpp: the file was added. It contains algorithm maintainance functions, which are explained elsewhere.

To give the reader some idea about why all this was necessary, the raw line counts of the files are presented before and after the modifications. While this is not an exact measurement, it indicates that the old interfaces demanded a more code to do the same thing.

```
             PICSearch 3.0   PICSearch 3.1
files under
server/inc      1011 lines       410 lines


files under
server/src      2687 lines      2086 lines
```

There has been an increase in functionality and an addition of a new file, Maint.cpp, which has 375 lines.

4.2.2 Client Modifications

Files that were touched in the client directory:

- inc/PICClient.h: Several function declarations in this file were modified. The modifications include the following:

  - xdrTranslate function replaced ReadServerMessage.

- AddIds, Login, AddDatabase, AddNewDatabase and RemoveDatabase were removed as unnecessary.
- AddGeneralOptions, ReceiveInit, AlgorithmProgressReport, QueryProgressReport, QueryResults, PictureData, UserDatam NewUserRights, NewDatabase, ServerOptions and NewPictureId were modified to take corresponding message parameters instead of generic PICMessage parameters.

- src/PICClient.cpp: Several functions were changed to use the new communication system. A list follows:
  - xdrTranslate replaced ReadServerMessage.
  - Login, AddDatabase, AddNewDatabase and RemoveDatabase were removed as unnecessary.
  - Connect, HandleConnection, Shutdown, ReadImageFromDb, InsertImageToDb, InsertPicture, GetPicture, DeletePicture, AddRemoveAlgorithms, CreateDatabase, DeleteDatabase, GetServerOptions, SetServerOptions, ReceiveInit, AlgorithmProgressReport, NewDatabase and ServerOptions were all more or less totally rewritten to use the new communication system.
- src/PICUser.cpp: A number of functions were modified, including:
  - UserData, NewUserRights, GetUsers, AddUser, RemoveUser, ChangePassword and AcceptRights were all heavily modified to utilize the new comm system.
- src/PICQuery.cpp: A lot of modifications, here, too.
  - AddIds was removed as unnecessary.
  - The two surviving StartQuerys, AddGeneralOptions, QueryProgressReport, QueryResults, PictureData, NewPictureId, HaltQuery and GetMoreThumbNails were all changed to use the new system.

Some stats:

```
             PICSearch 3.0   PICSearch 3.1
files under
client/inc      1041 lines       947 lines


files under
client/src      3125 lines      2838 lines
```

Again, functionality has strictly increased.

5 Graphical User Interface (TI)

The graphical user interface (GUI) of the PICSearch program is implemented with Tcl/Tk script language. These scripts communicate with the PicGUI module of the C++ implementation via an application specific Tcl command **transmit.** Once a **transmit** command is invoked with a desired C++ module service function as its first parameter, the PicGUI module searches an array of function pointers for a matching function. When one is found and **transmit** has received a correct number of additional arguments, the pointed function is invoked with the additional arguments received with the **transmit** call from Tcl/Tk. If the parameters are not correct, an error message is returned to the GUI. More specific information of the PICSearch GUI, and its version history, can be found from previous documents [3] [4] [5] of the previous projects.

5.1 New Elements of the GUI

The pics99 project added an easier way of including new search algorithms to the PICSearch program. This entry method is designed to be used through the PIC-Search client GUI. To give the user access to the necessary algorithm management functions, following windows were added to the GUI: *Add Algorithms, Modify Algorithms, Remove Algorithms*

Add Algorithms and *Modify Algorithms* offer user a way to add and to modify algorithms registered to a server. These windows were implemented mainly by using previously defined window construction procedures in file **littlehelper.tk**. Both of these windows are defined using same procedure with a parameter indicating exact functionality desired. Event handlers of these windows are also shared.

Remove Algorithms window is implemented in separate procedure and has its own event handler procedures. It allows user to remove algorithms from a server. Constructions procedures defined in **littlehelper.tk** were used for this window.

5.2 Enhanced Functionality of the GUI

New algorithm entry method also requires the user to have a way of activating algorithms in a desired database after the algorithms have been added to a server. For this task the existing window *Database Management* was altered to show all algorithms currently available in a server instead of only displaying those algorithms available in the current database. These are now shown as selected entries of the window. User activates an algorithm in a database simply by selecting it in addition of previously selected (ie. activated) algorithms. Activation (or deactivation) is confirmed by closing window with OK button.

New functionality added to the GUI is fully supported by, and totally compliant to the previous basic structures of the GUI. No new variable traces or interfaces to the C++ modules were needed.

Some previous tool procedures were too specific to suit the new demands in window construction. Rather than to add specific new tool procedures, or to enlarge existing ones, all needed additions were made by using basic TK script commands in relevant new procedures. Thus the new procedures themselves encapsulate all of their low level Tk operations not supported by previous tool procedures.

5.3 New Procedures and Methods

Following is a listing of entirely new Tcl/Tk procedures and C++ methods introduced in the GUI implementation of the **PICSearch v3.1**:

algorithmRemove Tcl/Tk, no parameters. Creates Remove Algorithms -window. File **topwindows.tk**.

algoRemoveOk Tcl/Tk, no parameters. Processes input when user has accepted an algorithm removal. File **topwindows.tk**.

algorithAddModify Tcl/Tk, parameter mode. Creates Add Algorithms or Modify Algorithms -window depending on received parameter. File **topwindows.tk**.

algoAddModOk Tcl/Tk, parameter mode. Processes input when user has accepted add or modify operation. Actual operation depends on received parameter. File **topwindows.tk**.

void PicGUI::addAlgorithm C++, parameter char *argv[], containing library file name, parameter window script file name and search type of the algorithm to be added. Packs received parameters to an internal message struct and sends them to the server for processing.

void PicGUI::replaceAlgorithm C++, parameter char *argv[], containing library file name, parameter window script file name and search type of the new implementation of the algorithm to be replace, id of the target algorithm and a flag indicating whether signatures for the target algorithm should be recalculated. Packs received parameters to an internal message struct and sends them to the server for processing.

void PicGUI::removeAlgorithm C++, parameter char *argv[], containing id of the algorithm to be removed. Sends id to the server for processing.

6 Algorithm Libraries (JL)

Every search algorithm in **PICSearch v3.1** is an external library, linked to the server at run-time. The algorithms use dlopen()- interface.

The algorithms from previous PICSearch version have been converted to external libraries. The new algorithms reside in "source/library/" directory. Old algorithms have been removed from "source/alg/".

A bare-bones no-operation "Skeleton" algorithm library has been included to the build to give a clean framework for building algorithms upon.

6.1 Changes to the Client

Support for old, fixed algorithms has been removed from the client. All knowledge about algorithms now originates from the server upon connection.

Client has been modified to support online algorithm maintenance. This mainly consist of receiving data from the GUI and sending it to the server. For more specific information, take a look at the GUI and XDR parts of this document.

6.2 Changes to the Server

- source/alg/inc/PicSignature.hxx (class PicSignature)

  The PicSignature class has undergone a severe simplification from the previous complex definition. It's now more like a struct than a class. It features only three public fields, and no functions.

  ```
  char *signature_data;            // Pointer to sig data
  int signature_size;              // Size of the data
  int creator_id;                  // ID of the algorithm
  ```

  NOTE: On delete, the signature tries to free "signature_data" if its not NULL. Make sure when using that it's either ok to free it, or set it to NULL in advance.

  PicSignature class is no longer inherited from anything.

- source/alg/inc/PicSearchAlgorithm.hxx (class PicSearchAlgorithm)

  Every algorithm library is seen as an object of PicSearchAlgorithm class, and the library functions are used just like normal member functions of an object.

The virtual functions getParams(), compare(), computeSignature(), algInfo() and algName() have been changed to function pointers for the library interface.

The only constructor is

```
PicSearchAlgorithm(char *lib_path, PICULONG id)
```

taking as its parameters the path to the library file, and an unique ID to be assigned to the algorithm. The database should keep count of the used IDs. The constructor tries to open the library, assign the function pointers and do some validity checking.

A function algID() has been been added to return the unique ID of the algorithm. If the initialization of an algorithm fails, the ID will be zero.

NOTE: computeSignature() function allocates memory for the signature. The caller must delete the signature when its no longer used.

PicSearchAlgorithm class is no longer inherited from anything.

- smain/src/maininit.cxx

  The initialization of the old, fixed algorithms has been removed. The new init is located in PicServer::Initialize(), in "source/server/src/Server.cpp".

- smain/src/mmisc.cxx

  Functions calcAll() and getSearchAlg() have been modified to accommodate the new libraries.

- server/*.cpp

  The database functions have been modified to understand the new signature class.

Totally removed files:

- smain/src/mainalgs.cxx
- smain/src/mainpics.cxx

7 Miscellaneous Changes (JL)

Amidst the horrible slaughter of innocent functions, several of the poor fellows may have gone missing in the excitement of the bloodshed, without being properly epitaphed to this mortuary. However, they all have been condemned as obsolete and unnecessary, and you shouldn't feel sad for their sorry fate.

Although the PICSearch cleaning squad tried to exterminate as much as they could, some twerps may have been left hiding in dark corners of the code. Please destroy encountered unnecessary functions on sight.

Here's some of the major miscellaneous changes.

7.1 Client

- source/client/src/PICClient.cpp

  The client of **PICSearch v3.0** featured an annoying time- trigger function which sent the query process reports to the GUI. The function always waited a fixed time limit before checking for a query progress report from the network socket. This was a bottleneck for query speed, because the server sent reports much faster than the client bothered to process them.

  The time-trigger function has been changed into Tcl_CreateFileHandler(), which waits upon the net socket for something to be readable.

  Query process is now just as fast as the server and network can provide.

- source/gui/src/PicGUI.cxx
  source/gui/src/*.tk
  source/cmain/inc/mshared.hxx
  source/cmain/src/maininit.cpp

  Removed "Insert Pictures to Database Physically" button. It wasn't connected in the first place, and wouldn't have much meaning in PICSearch operating over a network. Pictures have been inserted to the database physically since v3.0.

7.2 Changes Common to Both

- source/common/inc/PicList.hxx

  Because the PicSearchAlgorithm class is no longer part of the artificial PicObject class hierarchy, a new list class was needed to keep the algorithms in.

  A stardard STL list was considered and tried, but it didn't work in the antique compiling environment of University of Helsinki, dept. of Computer Science, and thus we couldn't use it.

A very simple PicList class was then implemented, which keeps anything in it that can be casted into (void *).

PicList provides the following member functions:

```
void listClean(void);          // Cleans this list (-> list will be empty)
void listRewind(void);         // Rewinds this list
void *listCurrent(void);       // Returns pointer to the current item
void *listNext(void);          // Advances position, returns next
int listAdd(void *);           // Adds a new item to the list
int listRemove(void *);        // Removes the item pointed to
```

NOTES:

- When listNext() returns NULL, the list has ended and needs to be rewinded with listRewind() before used again.

- listClean(), listRemove() and deleting the list object also free the respective items stored into the list.

- source/alg/inc/PicPicture.hxx (class PicPicture)

  A new field was added to make the ID based query possible.

  ```
  int DBid; // ID of the database containing this picture, 0=none
  ```

  This field should be initially be set to zero, and re-set to correct value if the picture is fetched from the database, or if its inserted to one and given an ID.

- Query options

  Query options (maximum matches, min match percentage) in the client have been fixed and should now work.

8 Testing of GUI Elements and Operations (TI)

The PICSearch GUI was tested independently of any functionality of the main program. This is possible because the GUI is implemented in Tcl/Tk scripts, which only provide operations for screen interaction and are not dependent from the actual C++ code of the main program. As there is only one connection between operational C++ parts of the code and Tcl/Tk window building and operating scripts it was simple to provide all the necessary testing feedback with minimal stubs implemented in PicGUI.cxx only for testing purposes. These stubs have been removed.

8.1 Windows in Menu Entry Order

Table 1 describes tests done to the different windows of the PICSearch program. All windows were tested for basic operations, namely invoke window, re-invoke while window already open, close with accepting (i.e. OK-button), close with no change (i.e. Cancel-button, also window close operation). Comments may contain additional observations about behavior or nature of a certain window. Result of "ok, proceeds" for an accepting close operation indicates that operations performed in PICSearch program through that window proceed normally.

window name	**invoke**	**re-invoke**	**close (ok)**	**close (cancel)**	**comments**
Connect	ok	ok, raises	ok, proceeds	ok, removed	all entry fields function ok
Database Management	ok	ok, raises	ok, proceeds	ok, removed	-
User Management	ok	ok, raises	ok, proceeds	ok, removed	-
Add Algorithms	ok	ok, raises	ok, proceeds	ok, removed	entry field values not initialized but "remembered"
Modify Algorithms	ok	ok, raises	ok, proceeds	ok, removed	entry field values not initialized but "remembered"
Select File	ok	modal	ok, proceeds	ok, removed	uses Tk file operation window
Remove Algorithms	ok	fixed, (a missing return statement)	ok, proceeds	ok, removed	-
Change Password	ok	modal, fixed (was not modal)	ok, proceeds	ok, removed	-
Batch Insert	ok	modal	ok, proceeds	ok, removed	uses Tk file operation window
Open Query Image	ok	modal	ok, proceeds	ok, removed	uses Tk file operation window
Save Query Image	ok	modal	ok, proceeds	ok, removed	uses Tk file operation window
Image Attributes	ok	ok, raises	ok, proceeds	ok, removed	-
Client Options	ok	ok, raises	ok, proceeds	ok, removed	unconnected features!
Server Options	ok	ok, raises	ok, proceeds	ok, removed	-
Server Advertisement	ok	modal	ok, closed	ok, closed	-
About PICSearch	ok	modal	ok, closed	ok, closed	-

Table 1: Windows of the PICSearch program

8.2 Operations in Menu Entry Order

Table 2 describes behavior of the selectable, non-windowed operations available in the PICSearch GUI. A status of "ok, proceeds" means that the operation specified is being carried out correctly upon menu selection. Conditions may specify some additional conditions for operation availability. These conditions have been tested to be system enforced.

operation name	connected	unconnected	other conditions	comments
Disconnect	ok, proceeds	N/A	-	-
Quit	ok, proceeds	ok, proceeds	-	-
Insert Query Image into D...	ok, proceeds	N/A	an image must be open and a db selected	-
Delete Selected Result I...	ok, proceeds(?)	N/A	an image must be selected from results	-
Available Images	ok, proceeds	N/A	-	tear-off window operates correctly
Close Query Image	ok, proceeds	N/A	a query image must be open	-
Get Selected Thumbnail	ok, proceeds(?)	N/A	an image must be selected from results	-
Get Selected Image	ok, proceeds(?)	N/A	an image must be selected from results	-
Image Editor	untested	N/A	-	Not set up correctly
On Selected Algorithm	ok, proceeds	N/A	an algorithm must be selected	-

Table 2: Windows of the PICSearch program

9 Algorithm Library Testing (JL)

Some testing has been done to ensure the correctness of the new algorithm library interface. The operations of converted example algorithms have also been tested.

9.1 Library Assigning Related

Library assigning is what server does upon initialization. Server finds algorithms in use from the database and tries to find the actual library files to assign the function pointers to.

Server performs the same operations during algorithm maintenance.

See Table 3.

Test case	Test result
Library file given in a maintenance operation does not exist (anymore?) on the file system.	Recognized, server does not try to use this library.
Library file is not a properly compiled dlopen()-interface compatible library.	Recognized, library not used.
Library format seems ok, but the necessary functions are not provided.	Recognized, library not used.
Library provides necessary functions, everything seems to be ok.	Ok, library is used.

Table 3: Library assignment tests

9.2 Library Using Related

There are several possibilities for the basic library operations to fail. The cases specified in Table 4 have been tested.

Test case	Test result
Library does something naughty, like segmentation violations.	Server dies. Host remains alive to spawn new servers. There's no way around this. Algorithm designers need to be careful.
computeSignature() returns a NULL signature.	Noticed, signature skipped.
Library returns invalid signature with size or content mismatch.	Server can't recognize good signatures from bad. If segfault happens because of this, server dies, host remains alive. Otherwise the bad signature is stored into the database.

Table 4: Tests related to the library use

9.3 Example Library Tests

For "Wavelet" and "AverageColor" algorithm libraries the items shown in Table 5 have been tested.

Test case	**Test result**
No raw picture data given for computeSignature()	Returns NULL signature.
Signatures given for compare() do not seem to be valid.	Zero match percentage returned.
Signature memory allocation error in computeSignature()	Returns NULL signature.
Correct information given for computeSignature() and compare()	Ok.
Use of algName() and algInfo()	Ok.
Use of setParams()	Wavelet has no parameters. Average-Color works ok. Invalid parameters are detected and not used.

Table 5: Example library tests

9.4 General Tests

Some crucial parts of the client might have been affected by the new algorithm library interfaces. See Table 6.

Test case	**Test result**
Query image insertion to database	Working
Mass insertion to database	Working
Query using a query image already in database	Was broken. Fixed.
Query with image not in database	Working
Query based on picture ID	Was broken. Fixed.
Query with thumbnail	Was broken. Fixed.
Saving of thumbnail	Was broken. Fixed.
Information on selected algorithm	Working

Table 6: General tests

10 Testing of Internal Communication

As internal communication of PICSearch is now built up on XDR, it is relatively free of operational defect, ie. it will always deliver messages, and messages themselves will be error-free. Continuous module testing of various parts of internal communication have exposed several error, which have been corrected. However, no robust system testing and very little integration testing has been done due to the severe time limits.

11 Database testing

The GDBM database manager now in place is itself a well-known and tested implementation. Various PICSearch functions utilizing database have been continuously module tested. Due to the pressing time limits, no full integration testing or good coverage system testing has been undertaken, therefore implementation of various database interface methods is not guaranteed to be error-free.

12 Performance (JL)

No formal performance testing of **PICSearch v3.1** has been done.

However, the following can be said:

- In query performance, v3.1 is now visibly faster on an average 586-class processor than v3.0, using any picture database exceeding 10 images. v3.1 also scales to larger databases.
- This version also seems much more stable and less buggy than its predecessor. Considerable amount of time was spent on bug-hunting and streamlining. The result is naturally a significant usability enhancement for the end-user, although quirks still remain.

Generally it can be claimed that the performance has increased where it counts.

13 Maintenance (JL)

During the three first phases of PICSearch development the software had quite well adapted to the so called ”mafia principle”, meaning that if you mess with one part of the program, the others will most certainly come to ”pay their respects” later.

Although the fourth team tried to make PICSearch a lot cleaner and simpler, it may still be rather hard to get the necessary big picture of the software in order to maintain it.

The worst problem is the lack of up-to-date documentation, clearly explaining the current state of the software, not just the changes from the previous versions. Reading all the documentations produced by the different development teams will most probably leave the reader in a confused state and not much the wiser.

13.1 Directory structure

At the moment there is no shortcut to mastering PICSearch, and the best way for understanding the internal organs of the software is to browse through the code. A quick help to the directory structure can be provided here, though. See Table 7.

Directory	**Belongs to**	**Contains**
alg/	both	Stuff related to the picture formats and their run-time representation.
client/	client	Primary parts belonging to the client
cmain/	client	Client main() part, misc client junk
common/	both	The heart of darkness, inherited code and general classes from version 1. Lots of redundant material
formats/	both	Currently the JPEG library
gui/	client	Tk GUI scripts and the C++ interface.
host/	server	The host which spawns the servers
library/	server	The search algorithm libraries
net/	both	Basic socket initialization routines
newdb/	server	v3.1 database routines
server/	server	Primary parts belonging to the server
smain/	server	Server main() part, misc server junk
xdr/	both	XDR communication

Table 7: PICSearch source/ directory structure

13.2 Miscellaneous hints

- Notice that the search algorithm libraries usually depend on classes defined in the main program, atleast on the PicPicture class. If you change those classes and do not update your algorithms, they will produce incorrect results.

Glossary

Following terms have been used in this document. Some meanings may be project-specific.

algorithm

In this document, algorithm references to a content-based image comparing algorithm. Basically, these algorithms just decide how closely two pictures resemble each other. PICSearch uses these to compare a user-provided query image with the pictures in the database.

bug

An error in the source code of a program. A bug is generally considered to be an error which does not stop the program from being compiled, but instead causes it to function erratically, stop functioning, or to function in ways not considered appropriate.

dead code

A useless section of the source code of a program, ie. a section which does not get executed when program is run, or which duplicates operations already implemented in some other code section.

debugging

Removing errors (see: bug) from the source code of a program.

gutted

In this document, gutting means the removal of operative code, while the function definition is left, returning a simple value in the fashion of "boolTrue".

hard coded

A value used by a program in its calculations is said to be hard coded if it is explicitly embedded in the source code. These values typically include some numerical limitations of the program, for example maximum number of permitted users could be hard coded as 99, as opposed to using a constant MAX_USERS_PERMITTED or some variable.

signature

Every algorithm produces a signature out of a source image. The signature contains those things the algorithm thinks important in the source image. Signatures of two different pictures are used to check for resemblance, not the pictures themselves.

References

[1]
K. Lemström, J. Korte, P. Kuusi, P. Kyheröinen and P. Päiväkumpu:
PICSearch - A platform for Image Content-based Searching Algorithms, (Technical Report C-1997-48), proceedings of the Sixth International Conference in Central Europe on Computer Graphics and Visualisation 98 (WSCG'98), edited by Vaclav Skala, Chzech Republic, February 9–13, 1998, pp. 222–229.
`http://www.cs.helsinki.fi/ klemstro/publications.html`

[2]
P. Kuusi, K. Lemström, P. Kanerva, S. Koponen, S. Männynsalo, J. Riihimäki, J. Teinilä and T. Virtanen:
A Client-Server Tool for Image Retrieval Applications, In: Proceedings of the HeCSE Workshop on Emerging Technologies in Distributed Systems, Helsinki University of Technology, Department of Computer Science and Engineering, pp. 45–46 (Research Report A50). Otaniemi, Finland. March 1998.

[3]
S. Kaski, T. Koskinen, J. Leinonen and M. Pennannen:
pics98, Implementation Specification, Helsinki, Finland, September 10, 1998.
`http://www.cs.helsinki.fi/group/pics98/`

[4]
J. Korte, P. Kuusi, P. Kyheröinen ja P. Päiväkumpu:
PICSearch-projektin (PICSearch v1) vaatimusmäärittely, Teknillinen korkeakoulu, 1996.

[5]
P. Kanerva, S. Koponen, S. Männynsalo, J. Riihimäki, J. Teinilä ja T. Virtanen:
NetPICSearch-projektin (PICSearch v2) vaatimusmäärittely, Helsingin yliopisto, Tietojenkäsittelytieteen laitos, 1997.
`http://www.cs.helsinki.fi/group/netpics/VM/VaatM12.html`

[6]
S. Kaski, T. Koskinen, J. Leinonen ja M. Pennannen:
pics98, Vaatimusmäärittely, Helsingin yliopisto, Tietojenkäsittelytieteen laitos, 1997.
`http://www.cs.helsinki.fi/group/pics98/`