BIOMETRIC EVALUATION COMMON FRAMEWORK

PROGRAMMER'S GUIDE
VERSION 0.1

WAYNE SALAMON
GREGORY FIUMARA

IMAGE GROUP
INFORMATION ACCESS DIVISION
INFORMATION TECHNOLOGY LABORATORY

National Institute of Standards and Technology
U.S. Department of Commerce

JANUARY 14, 2015

Contents

Chapter 1

Introduction

This document describes the Biometric Evaluation Framework (BECommon) and application programming interfaces (API) used to support the evaluation of biometric software within the NIST Image Group [14].

When evaluating software in a "black box" fashion many aspects of program execution must be addressed, such as non-returning function calls, I/O errors, and other resource requirements. In addition, solutions to common problems should be portable across operating systems.

An evaluation consists of the testing of vendor-supplied software that implements certain biometric algorithms, such as fingerprint matching or face recognition. The NIST Image Group defines a test process and API for each evaluation. Vendors implement the API in their software, which is delivered to NIST as a software library, where common test driver is used to call the vendor library. In order to support the common functionality used across all evaluations, such as logging, file input/output, etc., a common framework is used.

Even though the Biometric Evaluation Framework was written to support biometric software evaluations, much of the framework can be used for any general purpose program where data storage and system interaction are needed. One goal of the BECommon is to reduce the low-level error processing (particularly with input and output) done directly by applications. The Biometric Evaluation Framework provides several abstractions that are useful to applications so they can focus on the task at hand.

This document describes each package and includes example code. The long form of this document includes reference sections containing auto-generated API documentation.

The BECommon is a work-in-progress, and future development will occur in areas where the need arises for the testing programs of the NIST Image Group.

Chapter 2

Overview

The Biometric Evaluation Framework (BECommon) is a set of C++[16] classes, error codes, and design patterns used to create a common environment to provide logging, data management, error handling, and other functionality that is needed for many applications used in the testing of biometric software. The goals of the framework include:

- Reduce the amount of I/O error handling implemented by applications.
- Provide standard interfaces for data management and logging;
- Remove the need for applications to handle low-level events from the operating system (signals, etc.);
- Provide services for timing the execution of code blocks;
- Allow applications to constrain the amount of processing time used by a block of code;
- Reduce memory allocation errors;
- Simplify the use of parallel processing.

The experience of the NIST Image Group when running many software evaluations has led to the need of a common code for dealing with recurring software issues. One issue is the large amounts of data consumed, and created, by the software under test. Input data sets are typically biometric images, while output sets contain derived information. Both sets of data often contain millions of items, and storing each item as a file creates a tremendous burden on the file system. The `IO` package provides a solution to managing large amounts of records in a portable, efficient manner, as well as facilities for logging and maintaining runtime settings.

BECommon is divided into several packages, each providing a set of related functionality, such as error handling and timing operations. The packages are an informal concept, mapped to formal C++ name spaces, e.g. `IO` and `Time`. A namespace contains classes, constants, and non-class functions that relate to concepts grouped in the namespace. All classes within BECommon belong to the top-level `BiometricEvaluation` namespace.

Biometric image data is often supplied in a compressed format (e.g. WSQ, JPEG) and must be converted to a "raw" format. The `Image` package contains classes to represent compressed image data as an object, storing the image size and other attributes, in addition to the raw image.

Memory management issues are addressed by the `Memory` package. The use of classes and templates in this package can relieve applications of the need to directly manage memory for dynamically sized arrays, or call functions that are already provided to allocate and free C library objects.

While a program is running, it is often necessary to record certain statistics about the process, such as memory and processor usage. The `Process` package provides methods to obtain this information, as well as the capability to log to a file periodically, in an asynchronous manner.

In addition to its own statistics, a program may need to query some information about the environment under which it is running. The `System` package provides a count of CPUs, memory size, other system characteristics that an application can use to tailor its behavior.

Many aspects of software performance evaluation involve the use of timers. The `Time` package provides for the calculation of a time interval in a manner that is consistent across platforms, abstracting the underlying operating system's timing facility. Also, included is a "watchdog" facility, providing a solution to the problem of non-returning function calls. By using a watchdog timer, an application can abort a call to a function that doesn't return in the required interval.

The `Text` package provides a set of utility functions for operating on strings. The `digest` functions are of interest to those applications that must mask any information contained in a string before passing that information to another function. For example, often the biometric image file (or record) names contain information about the image, such as the finger position.

Error propagation and handling are addressed by the `Error` package. A set of exception objects are defined within this package, allowing for communication of error conditions out of the framework to the application, along with an explanatory string. Signal handling is related to error propagation in that when a process receives a signal, often it is due to software bug. Divide by zero, for example. The `Error` package provides for simple handling of the signal by the process.

Many packages in BECommon deal with biometric data record formats, including ANSI/NIST [3] records. In order to provide a general interface to several formats, BECommon represents the biometric data as derived from a source. For example, the `Finger` package contains classes that represent all information about a finger, including the source image and derived minutiae points. The `View` package combines the notions of a source image and derived information together into a single abstraction.

Applications can use the `Messaging` package to communicate between threads and processes, or to a terminal. Messages in this context are simply an array of bytes. One such use could be providing a command line interface to an long-running process.

The `MPI` package provides wrappers around the Message Passing Interface (MPI) [12] libraries, handling all MPI communcation and error events. Many parallel applications can be greatly simplified, only implementing a few methods to process data.

BECommon is designed to be used in a modular fashion, and it is possible to compile many packages independently. However, several packages do make use of other packages in the framework, and therefore, are less flexible in their reuse. However, BECommon is designed to reduce the intra-framework dependencies.

A set of test programs is included with the framework. These programs not only exercise the functions provided by the packages, but also can be used as example programs on how to use framework.

The chapters that follow this overview describe each package in detail, along with some code examples. The final set of chapters of this document contain the application programming interfaces for the types, methods, and classes that make up BECommon. However, the framework is under development, and other packages, classes, etc. will be added over time to address the needs of the NIST Image Group.

Chapter 3

Framework

The `Framework` package is used to retrieve information about the Biometric Evaluation Framework itself, as well as to provide services through general purpose utility functions to other parts of the framework.

3.1 Versioning

Version numbers, the compiler used, and other framework metadata can be queried by applications. Versioning information is recorded in the BECommon `Makefile` and populated in the function implentation at compile-time.

Listing 3.1: Using the `Framework` API

```
/* "Framework Version: 0.4" */
std::cout << "Framework Version: " << BE::Framework::getMajorVersion() << "." <<
    BE::Framework::getMinorVersion() << std::endl;

/* "Compiler Used: clang v5.1.0" */
std::cout << "Compiler Used: " << BE::Framework::getCompiler() << " v" <<
    BE::Framework::getCompilerVersion() << std::endl;

/* "Date/Time Compiled: Jan 24 2014 12:16:01" */
std::cout << "Date/Time Compiled: " << BE::Framework::getCompileDate() << " " <<
    BE::Framework::getCompileTime() << std::endl;
```

3.2 Enumerations

As of C++ 2011, `enum` s can be strongly-typed. The Biometric Evaluation Framework makes use of these strongly-typed `enum class` es throughout. As an added convenience, functions converting to and from `enum` s, `string` s, and `int` s are implicitly implemented easily via a `template`, eliminating many lines of boiler-plate code and creating equivalence in functionality among `enum class` es throughout BECommon.

At the core of `Framework::Enumeration` is a `const` mapping of `enum` to `string`, defined by you in code and instantated at compile-time. As demonstrated in Listing 3.2, simply define your `enum class` and populate the `map`.

Listing 3.2: `Framework::Enumeration`

```
/*
 * color.h
```

```
 */

enum class Color
{
        Black,
        Blue,
        Green
};

/*
 * color.cpp
 */

#include <be_framework_enumeration.h>

template<>
const std::map<Color, std::string>
BiometricEvaluation::Framework::EnumerationFunctions<Color>::enumToStringMap {
        {Color::Black, "Black"},
        {Color::Black, "Blue"},
        {Color::Green, "Green"}
};

/*
 * application.cpp
 */

#include <color.h>

/* "Black" */
std::cout << to_string(Color::Black) << std::endl;
/* "2" */
std::cout << to_int_type(Color::Green) << std::endl;
/* Color::Blue */
Color color = to_enum<Color>("Blue");
```

While `Framework::Enumeration` was created for BECommon, the `template`'s only dependency is `Exception`, and so it can easily be used in other C++ 2011 projects.

Chapter 4

Memory

To assist applications with memory management, the `Memory` package provides classes to wrap C memory allocations, and other dynamically-sized objects.

4.1 AutoBuffer

The Biometric Evaluation Framework is designed to interoperate with existing C code that has its own memory management techniques, e.g. NIST Biometric Image Software [13]. In these cases, functions exist to allocate and free blocks of memory, and these calls must be made by the applications which use those libraries. To assist BECommon clients that use these existing libraries, the `AutoBuffer` class wraps the C memory management functions, guaranteeing the release of C objects when the AutoBuffer goes out of scope.

The `AutoBuffer` constructor takes three function pointers as parameters: one for C object construction, one for destruction, and a third, optional, function for copying the C object. If the latter is passed a NULL, the `AutoBuffer` and the underlying C object cannot be copied, and an exception will be thrown.

Listing 4.1 shows the use of `AutoBuffer` to wrap the memory allocation routines that are part of the NIST Biometric Image Software ANSI/NIST library.

Listing 4.1: Using the `AutoBuffer`

```
#include <be_memory_autobuffer.h>
#include <iostream>
extern "C" {
  #include <an2k.h>
}

int
main(int argc, char* argv[]) {


    /*
     * alloc_ANSI_NIST(), free_ANSI_NIST(), and copy_ANSI_NIST()
     * are functions in the NBIS AN2K library.
     */
    Memory::AutoBuffer<ANSI_NIST> an2k =
        Memory::AutoBuffer<ANSI_NIST>(&alloc_ANSI_NIST,
            &free_ANSI_NIST, &copy_ANSI_NIST);
    if (read_ANSI_NIST(fp, an2k) != 0) {
            cerr << "Could not read AN2K file." << endl;
            return (EXIT_FAILURE);
```

```
    }

    for (int i = 1; i < an2k->num_records; i++) {
        // process the ANSI/NIST record ...
    }
}
```

4.2 AutoArray

At its simplest level, `AutoArray` is a C-style array with numerous convenience methods, such as being able to query the number of elements. C++ iterators can be used over the contents of the array. The array can be resized without the need to create a new object. C++ operator overloading allows `AutoArray` objects to be passed to C-style functions that expect pointers to `AutoArray`'s template type.

`AutoArray` is used extensively in BECommon to help eliminate mistakes when manually allocating memory. The `AutoArray` constructor will allocate needed memory using `new` and the destructor will `delete` it. This ensures that any allocated memory will be appropriately freed when the `AutoArray` goes out of scope. Copy constructors and methods as well as the assignment operator all correctly manage memory so the client does not have to. Several objects in BECommon `return AutoArray` objects to assist clients in proper memory management.

A common use of `AutoArray` is to deal with records sequenced from a `RecordStore`. Listing 4.2 demonstrates this. Notice the omission of memory management statements – they are completely unnecessary.

Listing 4.2: Using `AutoArray` s with `RecordStore` s

```
#include <be_io_dbrecstore.h>
#include <be_memory_autoarray.h>

#include <iostream>

using namespace BiometricEvaluation;

int
main(
    int argc,
    char *argv[])
{
        IO::DBRecordStore rs("db_recstore", ".", IO::READONLY);

        uint64_t value_size = 0;
        string key("");
        Memory::AutoArray<uint8_t> value;
        for (bool stop = false; stop == false; ) {
                try {
                        // Non-destructively resize the AutoArray to hold
                        // the next record.
                        value.resize(rs.sequence(key, NULL));

                        // Read the record into the AutoArray (treats the
                        // AutoArray as a pointer).
                        rs.read(key, value);

                        // Do something with value.
                        std::cout << "Key " << key << " has a value of " <<
                            value.size() << " bytes" << std::endl;
```

```
                } catch (Error::ObjectDoesNotExist) {
                        stop = true;
                }
        }

        return (0);
}
```

AutoArray is adapted from ”c_array ” [16, 496].

4.3 IndexedBuffer

Many applications have a need to read items from a data record and take action based on the value of the item read. For example, when reading a biometric data record, the number of finger minutiae points in the record is indicated by a value in the record header. Furthermore, the record format may be of a different endianess than the application's host platform.

The IndexedBuffer class is used to access data from a buffer in fixed-size amounts in sequence. Objects of this class maintain an index into the buffer as internal state and reads out of the buffer, when using certain methods, adjust the index. In addition, standard subscript access can be done on on the buffer (reads and writes) without affecting the index. The basic element type is an unsigned eight-bit value. The IndexedBuffer object can be created to either manage the buffer memory directly, or to "wrap" an existing buffer.

Methods to retrieve elements from the buffer are defined in the class's interface. These functions are used to retrieve 8/16/32/64-bit values while moving the internal index. Several functions are also provided to take into account the endianess of the underlying data.

Listing 4.3 shows how an application can read a data record in big-endian format.

Listing 4.3: Using the IndexedBuffer

```
#include <be_memory_autoarray.h>
#include <be_memory_indexedbuffer.h>

int
main(int argc, char* argv[]) {

        uint64_t size = IO::Utility::getFileSize("BiometricRecord");
        FILE *fp = std::fopen("BiometricRecord", "rb");
        Memory::IndexedBuffer iBuf(size);
        fread(iBuf, 1, size, fp);
        fclose(fp);
        Memory::IndexedBuffer iBuf(recordData, recordData.size());

        uint32_t lval;
        uint16_t sval;

        /*
         * Record is big-endian:
         * ---------------------------------
         * | NAME | LENGTH | ID |    ...    |
         * ---------------------------------
         *     4       4      2
         */

        /* Read a 4-byte C string */
        lval = iBuf.scanU32Val();          /* Format ID */
        char *cptr = (char *)&lval;
```

```
        string s(cptr);

        /* Read a 4-byte length */
        lval = iBuf.scanBeU32Val()

        /* Read a 2-byte ID */
        sval = iBuf.scanBeU16Val();
}
```

Chapter 5

Error Handling

Within the Biometric Evaluation Framework, Error handling has two aspects: One for communicating error conditions out of the framework and back to applications; the other for handling error signals from the environment and operating system. Classes and other code to implement error processing are described in this chapter.

5.1 Biometric Evaluation Exceptions

The Biometric Evaluation Framework contains a set of classes used to report errors to applications. Objects of these class types are thrown and contain descriptive information as to the nature of the error. Applications must handle the errors in a manner that makes sense for the application.

Applications should catch objects of the type specified in the API for the class being called. The type of object caught indicates the nature of the error that occurred, while the string stored within that object provides more information on the error.

Listing 6.2 on page 17 shows an example of exception handling when using the logging classes described in Section 6.3 on page 16.

5.2 Signal Handling

When the application process executes in a POSIX environment, signals to the process can be generated by the operating system. In many cases, if the signal is not handled by the process, execution terminates. Because the Biometric Evaluation Framework was designed to used with software libraries for which no source code is available, changes to the code in these libraries cannot be made, and any faults in that code cannot be fixed. A common problem is that a function in the "black box" library dereferences a bad pointer, resulting in a segmentation violation signal being sent by the operating system.

To prevent termination of the application process, signal handling must be installed. The Biometric Evaluation Framework provides a class, `SignalManager`, to simplify the installation of a signal handler in order to allow the program to continue running. For example, when extracting a fingerprint minutia template from an image, often the library call will fault on a certain image. By using the `SignalManager`, the application can log that fault, and continue on to the next image.

Signal handling in a POSIX environment covers the bare essentials, and one of two actions is usually taken. The signal can be handled and processing continues at the location the signal was generated. The second action is that, in addition to signal handling, the process continues from a different location. It is the second action that is implemented by the `SignalManager` class. The rationale for this type of signal handling is so the call to the faulting function can be aborted, but the caller can detect that the signal was handled and take action, usually by logging the fault.

By default, the `SignalManager` class installs a handler for the `SIGSEGV` and `SIGBUS` signals. However, other signals can be handled as desired.

One restriction on the use of `SignalManager` is that the POSIX calls for signal management (`signal(3)`, `sigaction(2)`, etc.) cannot be invoked inside of the signal handler block.

The example in Listing 5.1 shows application use of the `SignalManager` class.

Listing 5.1: Using the `SignalManger`

```
#include <be_error_signal_manager.h>
using namespace BiometricEvaluation;

int main(int argc, char *argv[])
{
        Error::SignalManager *sigmgr = new Error::SignalManager();

        BEGIN_SIGNAL_BLOCK(sigmgr, sigblock1);
        // code that may result in signal generation
        END_SIGNAL_BLOCK(asigmgr, sigblock1);
        if (sigmgr->sigHandled()) {
                // log the event, etc.
        }
}
```

Within the `SignalManager` header file, two macros are defined: `BEGIN_SIGNAL_BLOCK()` and `END_SIGNAL_BLOCK()`, each taking the `SignalManager` object and label as parameters. The label must be unique for each signal block. These macros insert the jump buffer into the code, which is the location where the signal handler will jump to after handling the signal. The use of these macros greatly simplifies signal handling for the application, and it is recommended that applications use these macros instead of directly invoking the methods of the `SignalManger` class, except for changing the set of handled signals.

If a signal does occur, process control jumps to the end of the signal block, and the `sigHandled()` method of the signal manager can be called. The application may need to have the same statements inside the `sigHandled()` check as those outside of the signal handling block. For example, if a file needs to be closed before the end of the block, the same call to the close function must be made within the `sigHandled()` check. Careful application design can reduce the amount of code replication, however.

Listing 5.2 shows how an application can indicate what signals to handle. In this example, only the `SIGUSR1` signal would be handled.

Listing 5.2: Specifying Signals to the `SignalManger`

```
#include <be_error_signal_manager.h>
using namespace BiometricEvaluation;

int main(int argc, char *argv[])
{
    Error::SignalManager *sigmgr = new Error::SignalManager();

    sigset_t sigset;
    sigemptyset(&sigset);
    sigaddset(&sigset, SIGUSR1);
    sigmgr->setSignalSet(sigset);

    FILE *fp = fopen( ... );
    BEGIN_SIGNAL_BLOCK(sigmgr, sigblock2);
        // code that may result in signal generation
        fclose(fp);
    END_SIGNAL_BLOCK(asigmgr, sigblock2);
```

```
    if (sigmgr->sigHandled()) {
        cout << "SIGUSR1 occurred." << endl;
        fclose(fp);
    }
}
```

Chapter 6

Input/Output

The `IO` package is used by applications for the common types of input and output: managing stores of data, log files, and individual file management. The goal of using the `IO` API is to relieve applications of the need to manage low-level I/O operations such as file opening, writing, and error handling. Furthermore, by using the classes defined in `IO`, the actual storage mechanism used for data can be managed efficiently and placed in a consistent location for all applications.

Many classes manage persistent storage within the file system, taking care of file open and close operations, as well as error handling. When errors do occur, exceptions are thrown, which then must be handled by the application.

6.1 Utility

The `IO::Utility` namespace provides functions that are used to manipulate the file system and other low-level mechanisms. These functions can be used by applications in addition to being used by other classes within the Biometric Evaluation framework. The functions in this package are used to directly manipulate objects in the POSIX file system, or to check whether a file object exists.

6.2 Record Management

The `IO::RecordStore` class provides an abstraction for performing record-oriented input and output to an underlying storage system. Each implementation of the `RecordStore` provides a self-contained entity to manage data on behalf of the application in a reliable, efficient manner.

Many biometric evaluations generate thousands of files in the form of processed images and biometric templates, in addition to consuming large numbers of files as input. In many file systems, managing large numbers of files in not efficient, and leads to longer run times as well as difficulty in backing up and processing these files outside of the actual evaluation.

The `RecordStore` abstraction de-couples the application from the underlying storage, enabling the implementation of different strategies for data management. One simple strategy is to store each record into a separate file, reproducing what has typically been done in the evaluation software itself. Archive files and small databases are other implementation strategies that have been used.

Use of the `RecordStore` abstraction allows applications to switch storage strategy by changing a few lines of code. Furthermore, error handling is consistent for all strategies by the use of common exceptions.

`RecordStore` s provide no semantic meaning to the nature of the data that passes through the store. Each record is an opaque object, given to the store as a pointer and data length, and is associated with a string the which is the key. Keys must be unique and are associated with a single record. Attempts to insert multiple records with the same key result in an exception being thrown.

Listing 6.1 illustrates the use of a database `RecordStore` within an application.

Listing 6.1: Using a `RecordStore`

```
#include <be_io_dbrecstore.h>
int
main(int argc, char* argv[]) {

    IO::DBRecordStore *rs;
    try {
        rs = new IO::DBRecordStore("myRecords", "My Record Store");
    } catch (Error::Exception& e) {
        cout << "Caught " << e.what() << endl;
        return (EXIT_FAILURE);
    }
    std::unique_ptr<IO::DBRecordStore> urs(rs);

    try {
        Memory::uint8Array theData;

        theData = getSomeData();
        urs->insert("key1", theData);

        theData = getSomeData();
        urs->insert("key2", theData);

    } catch (Error::Exception& e) {
        cout << "Caught " << e.what() << endl;
        return (EXIT_FAILURE);
    }

    // Some more processing where new data for a key comes in ...
    theData = getSomeData();
    urs->replace("key1", theData);

    // Obtain the data for all keys ...
    string theKey;
    while (true) {
        uint64_t len = rs->sequence(theKey, theData);
        cout << "Read data for key " << theKey << " of length " << len << endl;
    }
    // The data for the key is no longer needed ...
    urs->remove("key1");
}
```

6.3 Logging

Many applications are required to log information during their processing. In particular, the evaluation test drivers often create a log record for each call to the software under test. There is a need for the log entries to be consistent, yet any logging facility must be flexible in accepting the type of data that is to be written to the log file.

The logging classes in the `IO` package provide a straight-forward method for applications to record their progress without the need to manage the low-level storage details. Management of the log messages to the backing store is done within the `Logsheet` implementations. `Logsheet` specifies the common interface to all implementations. In addition, objects of this class can be created to provide a “Null” `Logsheet` where messages are not saved.

A `Logsheet` is an output stream (subclass of `std::ostringstream`), and therefore can handle built-in types and any class that supports streaming. Each entry is numbered by the `Logsheet` class when written to the log. A call to the `newEntry()` method commits the current entry to the log, and resets the write position to the beginning of the entry buffer.

In addition to streaming by using the `Logsheet::<<` operator, applications can directly commit an entry to the log file by calling the `write()` method, thereby not disrupting the entry that is being formed. After an entry is committed, the entry number is automatically incremented. `Logsheet` also supports the writing of “debug” and comment entries. Each entry is prefixed with a letter code indicating the type.

6.3.1 FileLogsheet

`IO::FileLogsheet` uses a file to store the log messages. Access to this file is not controlled, and therefore, if two instances of this class are made with the same file name, the results are undefined. The description of the sheet is placed at the top of the file during construction of the object. Objects of this class can be constructed with a string containing a `file://` Uniform Resource Locator (URL) or a simple file name.

`IO::FileLogCabinet` is a container of `FileLogsheet` where each log file is contained within the same directory owned by this container class.

The example code in Listing 6.2 shows how an application can use a `FileLogsheet`, contained within a `FileLogCabinet`, to record operational information.

Listing 6.2: Using a `FileLogsheet` within a `FileLogCabinet`

```
#include <be_io_filelogcabinet.h>
using namespace BiometricEvaluation;
using namespace BiometricEvaluation::IO;

FileLogCabinet *lc;
try {
    lc = new FileLogCabinet(lcname, "A Log Cabinet", "");
} catch (Error::ObjectExists &e) {
    cout << "The Log Cabinet already exists." << endl;
    return (-1);
} catch (Error::StrategyError& e) {
    cout << "Caught " << e.what() << endl;
    return (-1);
}
std::unique_ptr<FileLogCabinet> ulc(lc);
try {
    ls = alc->newLogsheet("log01", "Log Sheet in Cabinet");
} catch (Error::ObjectExists &e) {
    cout << "The log sheet already exists." << endl;
    return (-1);
} catch (Error::StrategyError& e) {
    cout << "Caught " << e.what() << endl;
    return (-1);
}
ls->setAutoSync(true);  // Force write of every entry when finished
int i = ...
*ls << "Adding an integer value " << i << " to the log." << endl;
ls->newEntry();         // Forces the write of the current entry
.........
delete ls;
return;                 // The LogCabinet is destructed by the unique_ptr
```

6.3.2 SysLogsheet

The `SysLogsheet` is an implementation of `Logsheet` which writes log entries to a system logger service. Objects of this class are created with a URL starting with `syslog://`. When using a system logger, the URL must give the hostname of the logger as well as the network port: `syslog://node00:4315` for example. The system logger must understand the Syslog protocol as specified in RFC5424 [17].

Multiple instances of a `SysLogsheet` can be created with the same URL with the assumption that the logging server can manage multiple incoming message streams.

6.4 Properties

The `Properties` class is used to store simple key-value string pairs, with the option to save to a file. Applications can use a `Properties` object to manage runtime settings that are persistent across invocations, or to simply store some settings in memory only.

Listing 6.3: Using a `Properties` Object

```
IO::Properties *props;
string fname = "test.prop";
try {
    props = new IO::Properties(fname);
} catch (Error::StrategyError &e) {
    cerr << "Caught " << e.what()  << endl;
    return;
} catch (Error::FileError& e) {
    cerr << "A file error occurred: " << e.what() << endl;
    return;
}
props->setProperty("foo", "bar");
props->setProperty("theAnswer", "42");
    :
    :
    :
try {
    int64_t theAnswer = props->getProperty("theAnswer");
    cout << "The answer is " << theAnswer << endl;
} catch (Error::ObjectDoesNotExist &e) {
    cerr << "The answer is elusive." << endl;
    return;
}
string fooProp = props->getProperty("foo");
cout << "Foo is set to " << fooProp << endl;
    :
    :
    :
try {
    props->removeProperty("foo");
} catch (Error::ObjectDoesNotExist &e) {
    cerr << "Failed to remove property." << endl;
}
```

6.5 Compressor

Support for data compression and decompression can be found in the Biometric Evaluation Framework through the `Compressor` class hierarchy. `Compressor` is an abstract base class defining several pure-virtual methods for compression and decompression of buffers and files. Derived classes implement these methods and can be instantiated through the factory method in the base class. As such, children should also be enumerated within `Compressor::Kind`. The Biometric Evaluation Framework comes with an example, `GZIP`, which compresses and decompresses the `gzip` format through interaction with `zlib` [4].

Listing 6.4: Using a `Compressor` Object

```
shared_ptr<IO::Compressor> compressor;
Memory::uint8Array compressedBuffer, largeBuffer = /* ... */;
try {
        compressor = IO::Compressor::createCompressor(Compressor::Kind::GZIP);
        /* Overloaded for all combination of buffer and file */
        compressor->compress("largeInputFile", "compressedOutputFile");
        compressor->compress(largeBuffer, compressedBuffer);
} catch (Error::Exception &e) {
        cerr << "Could not compress (" << e.what() << ')' << endl;
}
```

Different `Compressor` s may be able to respond to options that tune their operations. These options (and approved values) should be well-documented in the child class, however, a no-argument constructor of a child `Compressor` should automatically set any required options to default values. Setting and retrieving these options is very similar to interacting with a `Properties` object (see Section 6.4 on the facing page).

Listing 6.5: Setting `Compressor` Options

```
shared_ptr<IO::Compressor> compressor =
    IO::Compressor::createCompressor(Compressor::Kind::GZIP);

/* A large GZIP chunk size can speed operations on systems with copious RAM */
compressor->setOption(IO::GZIP::CHUNK_SIZE, 32768);
```

Chapter 7

Time and Timing

The `Time` package within the Biometric Evaluation Framework provides a set of classes for performing timing-related operations, such as elapsed time and limiting execution time.

7.1 Elapsed Time

The `Timer` class provides applications a method to determine how long a block of code takes to execute. On many systems (e.g. Linux) the timer resolution is in microseconds.

Listing 7.1 shows how an application can use a `Timer` object to limit obtain the amount of time used for the execution of a block of code.

Listing 7.1: Using the `Timer`

```
#include <be_time_timer.h>

int main(int argc, char *argv[])
{
	Time::Timer timer = new Time::Timer();

	try {
		atimer->start();
		// do something useful, or not
		atimer->stop();
		cout << "Elapsed time: " << atimer->elapsed() << endl;
	} catch (Error::StrategyError &e) {
		cout << "Failed to create timer." << endl;
	}
}
```

7.2 Limiting Execution Time

The `Watchdog` class allows applications to control the amount of time that a block of code has to execute. The time can be *real* (i.e. "wall") time, or *process* time (not available on Windows). One typical usage for a `Watchdog` timer is when a call is made to a function that may never return, due to problems processing an input biometric image.

`Watchdog` timers can be used in conjunction with `SignalManager` in order to both limit the processing time of a call, and handle all signals generated as a result of that call. See 5.2 for information on the `SignalManager` class.

One restriction on the use of `Watchdog` is that the POSIX calls for signal management (`signal(3)`, `sigaction(2)`, etc.) cannot be invoked inside of the `WATCHDOG` block. This restriction includes calls to `sleep(3)` because it is based on signal handling as well.

Listing 7.2 shows how an application can use a `Watchdog` object to limit the about of process time for a block of code.

Listing 7.2: Using the `Watchdog`

```
#include <be_time_watchdog.h>
int main(int argc, char *argv[])

    Time::Watchdog theDog = new Time::Watchdog(Time::Watchdog::PROCESSTIME);
    theDog->setInterval(300);   // 300 microseconds

    Time::Timer timer;

    BEGIN_WATCHDOG_BLOCK(theDog, watchdogblock1);
        timer.start();
        // Do something that may take more than 300 usecs
        timer.stop();
        cout << "Total time was " <<  timer.elapsed() << endl;
    END_WATCHDOG_BLOCK(theDog, watchdogblock1);
    if (theDog->expired()) {
        timer.stop();
        cerr << "That took too long." << endl;
    }
{
}
```

Within the `Watchdog` header file, two macros are defined: `BEGIN_WATCHDOG_BLOCK()` and `END_WATCHDOG_BLOCK()`, each taking the `Watchdog` object and label as parameters. The label must be unique for each `WATCHDOG` block. The use of these macros greatly simplifies `Watchdog` timers for the application, and it is recommended that applications use these macros instead of directly invoking the methods of the `Watchdog` class, except for setting the timeout value.

Any processing that is normally done at the end of the `WATCHDOG` block must also be done within the `expired()` check due to the fact that process control jumps to the end of the `WATCHDOG` block in the event of a timeout. A typical example is the use of the `Timer` object inside a `WATCHDOG` block, as the example in Listing 7.2 shows. In most cases, however, careful application design can remove the need for duplicate code. In the example, placing the `Timer start()/stop()` calls outside of the `WATCHDOG` block simplifies the coding, although the small amount of time for the `WATCHDOG` setup and tear down would be included in the time.

Chapter 8

Process Information and Control

The `Process` package is a set of APIs used to gather information on a process, limit the capabilities of a process, and to manage the life cycle of processes.

8.1 Process Statistics

When a application is running, there may be a need to obtain information of the process executing that application. The `Process` can be used by the application itself to gather statistics related to the current amount of memory being used, the number of threads, and other items. Biometric evaluation test drivers are linked against a third party library, and therefore, the application writer does not control the thread count or memory usage for much of the processing. Listing 8.1 shows how an application can use the `Statistics` API.

Listing 8.1: Gathering Process Statistics

```
#include <be_error_exception.h>
#include <be_process_statistics.h>
using namespace BiometricEvaluation;

int main(int argc, char *argv[])
{
    Process::Statistics stats;
    uint64_t userstart, userend;
    uint64_t systemstart, systemend;
    uint64_t diff;
    try {
        stats.getCPUTimes(&userstart, &systemstart);

        // Do some long processing....

        stats.getCPUTimes(&userend, &systemend);
        diff = userend - userstart;
        cout << "User time elapsed is " << diff << endl;
        diff = systemend - systemstart;
        cout << "System time elapsed is " << diff << endl;
    } catch (Error::Exception) {
        cout << "Caught " << e.getInfo() << endl;
    }

}
```

In addition to using the `Process` API to gather statistics to be returned from the function call, the API provides a means to have a “standard” set of statistics logged either synchronously or asynchronously to a `LogSheet` (See Section 6.3 on page 16) contained within a `LogCabinet`. Applications can start and stop logging at will to this `LogSheet`. Post-mortem analysis can then be done on the entries in the log. Listing 8.2 shows the use of logging.

The `LogSheet` will have a file name constructed from the process name (i.e. the application executable) and the process ID. An example `LogSheet` contains this information at the start:

```
Description: Statistics for test_be_process_statistics (PID 28370)
# Entry Usertime Systime RSS VMSize VMPeak VMData VMStack Threads
E0000000001 728889 6998 1788 57472 62612 31020 84 1
E0000000002 1300802 6998 1792 57472 62612 31020 84 1
```

The `Statistics` object creates the `LogSheet` with an appropriate description and comment entry with column headers. Each gathering of the statistics results in a single log entry.

Listing 8.2: Logging Process Statistics

```cpp
#include <be_error_exception.h>
#include <be_io_logcabinet.h>
#include <be_process_statistics.h>
using namespace BiometricEvaluation;

int main(int argc, char *argv[])
{
    IO::LogCabinet lc("statLogCabinet", "Cabinet for Statistics", "");

    Process::Statistics *logstats;
    try {
        logstats = new Process::Statistics(&lc);
    } catch (Error::Exception &e) {
        cout << "Caught " << e.getInfo() << endl;
        return (EXIT_FAILURE);
    }
    try {
        while (some_processing_to_do) {
            // Do the work
            // Synchronously log after the work is done.
            logstats->logStats();
        }
    } catch (Error::Exception &e) {
        cout << "Caught " << e.getInfo() << endl;
        delete logstats;
        return (EXIT_FAILURE);
    }

    // Set up asynchronous logging, every second
    try {
        logstats->startAutoLogging(1);
    } catch (Error::ObjectExists &e) {
        cout << "Caught " << e.getInfo() << endl;
        delete logstats;
        return (EXIT_FAILURE);
    }

    // Do some other work
```

```

    // Stop logging
    logstats->stopAutoLogging();
    delete logstats;
}
```

8.2 Process Management

During a biometric evaluation or other long-running CPU-bound task, it's beneficial to make efficient use of all the hardware available on the system. Applications can take advantage of a multi-core machine, for example. BECommon aims to simply this by abstracting the usage of process and thread creation to run multiple instances of the same function in parallel.

8.2.1 Manager

There are three class hierarchies involved in the abstraction. The `BiometricEvaluation::Process::Manager` classes control the technique of process manipulation that will be used. BECommon provides two example abstractions: `ForkManager` and `POSIXThreadManager`. When using `ForkManager`, new processes will be created with `fork(2)`, with mediated access to these new processes through the `Manager`. Likewise, `POSIXThreadManager` creates new POSIX threads. Because both of theses classes inherit from `Manager`, it is as trivial as changing the `Manager` object type to change how the workload is parallelized.

8.2.2 Worker

In the application using a `Manager`, a `Worker` subclass must be implemented. An example `Worker` is shown in Listing 8.3. The entry-point for a `Worker` is the `workerMain()` method, which must be implemented by the client application. Although `workerMain()` takes no arguments, data may be transmitted into the object through `WorkerController`'s (8.2.3) `setParameter()` method. Within the `Worker` instance, the parameters are then retrieved with `getParameter()` when provided with the unique parameter name.

A responsible worker performs its operations as fast as it can. However, at any given time, the manager may ask the worker to stop. It then becomes the *responsibility of the worker* to stop as soon as possible. The `Worker` is notified of the stop request through its `stopRequested()` method. Note that the manager does **not** force the worker to stop, though prolonged work or cleanup in the worker would likely produce undesired results in the client application. As such, a responsible worker checkpoints itself to prepare for premature stops requested by the manager. While it is important for a worker to stop as soon as possible after the request is received, it is also important not to leave work in an unsynchronized state. In Listing 8.3, notice how the `Employee` must continue the interaction with the `Customer` before a stop request is handled, even if the `Employee`'s shift has ended. Leaving the method before the `Customer`'s order has been delivered would leave the `Customer` object in an unsafe state (hungry).

Listing 8.3: A Responsible `Worker` Implementation

```
#include <cstdlib>
#include <tr1/memory>
#include <queue>

#include <restaurant.h>

#include <be_process_forkmanager.h>

using namespace std;
using namespace BiometricEvaluation;
```

```
using namespace Restaurant;

class ResponsibleEmployeeTask : public Process::Worker
{
public:
        int32_t
        workerMain()
        {
                int32_t status = EXIT_FAILURE;

                /* Retrieve objects assigned to this Task */
                tr1::shared_ptr<Employee> employee =
                    tr1::static_pointer_cast<Employee>(
                    this->getParameter("employee"));
                tr1::shared_ptr< queue<Customer*> > customers =
                    tr1::static_pointer_cast< queue<Customer*> >(
                    this->getParameter("customers")

                employee->clockIn();

                Customer *customer;
                /* Checkpoint after each customer */
                while (this->stopRequested() == false ||
                    employee->isShiftOver() == false) {
                        customer = customers->front();

                        if (customer != NULL) {
                                employee->takeOrder(customer);
                                employee->cookFood(customer);
                                employee->deliverOrder(customer);

                                customers->pop();
                        }
                }

                employee->settleCashDrawer();
                employee->clockOut();

                status = EXIT_SUCCESS;
                return (status);
        }
        ~ResponsibleEmployeeTask() {}
};
```

After a manager starts its workers, the manager has the option of waiting until all `Worker` s exit `workerMain()` before continuing code execution. If not waiting, there are several methods the manager can perform to keep track of the status of the workers. Even if not waiting for workers to return, a responsible manager will wait a reasonable amount of time for workers to `return` before application termination. An example of this reasonable waiting period can be seen in Listing 8.4 on the next page.

8.2.3 WorkerController

The final piece of the process management puzzle is the `WorkerController` hierarchy. This class decorates and mediates communication between the `Manager` and the `Worker`. `WorkerController` objects may only be instantiated by a `Manager` object. All communications to the `Worker` (e.g. `isWorking()`) should be delegated through the `WorkerController`. If defining a new `Manager`, note that the `Worker`

Controller may seem unnecessary for the parallelization technique being employed. It's true that some parallelization techniques may not require this "middle-man" approach, but others do. Do not be concerned if a WorkerController implementation ends up being nothing more than a "pass-thru" to the Worker.

Listing 8.4 is a continuation of Listing 8.3 on page 25 demonstraiting the use of Manager s and Worker Controller s.

Listing 8.4: Using Manager s and WorkerController s

```
int
main(
    int argc,
    char *argv[])
{
        static const uint32_t numEmployees = 3;
        int status = EXIT_FAILURE;

        tr1::shared_ptr<Process::Manager> shiftLeader(new Process::ForkManager);
        queue<Customer*> *customers = new queue<Customer*>();

        /* Create Employees (Workers/WorkerControllers) */
        tr1::shared_ptr<Process::WorkerController> employees[numEmployees];
        for (uint32_t i = 0; i < numEmployees; i++) {
                employees[i] = shiftLeader->addWorker(
                    tr1::shared_ptr<ResponsibleEmployeeTask>(
                    new ResponsibleEmployeeTask()));

                /* Assign employees to each Task */
                employees[i]->setParameter("employee",
                    tr1::shared_ptr<Employee>(new Employee()));
                employees[i]->setParameter("customers",
                    tr1::shared_ptr< queue<Customer*> >(customers);
        }

        /* Employees start serving customers while shift leader manages */
        shiftLeader->startWorkers(false);

        /* Customers enter the queue... */
        queue<Restaurant::AdministrativeTasks> adminTasks;
        adminTasks.push("Inventory");
        adminTasks.push("Customer Complaints");
        adminTasks.push("Clean Dining Room");

        while (shiftLeader->getNumActiveWorkers() != 0) {
                shiftLeader->doTask(adminTasks.front());
                adminTasks.pop();
        }

        /* ...end of the day */
        for (uint32_t i = 0; i < numEmployees; i++)
                if (employees[i]->isWorking())
                        shiftLeader->stopWorker(employees[i]);

        /*
         * Wait a reasonable amount of time before locking up for the night
         * (in this case, indefinitely).
         */
```

```
        while (shiftLeader->getNumActiveWorkers() > 0)
                sleep(1);

        shiftLeader->armAlarmAndExit();

        status = EXIT_SUCCESS;
        return (status);
}
```

8.2.4 Communications

Managers and workers may have a good reason to send and receive messages directly. A communications mechanism is built-in to the Process Management model to facilitate such communications. The type and content of the message is completely up to the client implementation, since messages are sent as `AutoArray` s. A manager does not directly send messages to a worker. This service is provided by the `WorkerController` (via `sendMessageToWorker()`).

Managers can keep an eye on incoming messages by calling the (optionally blocking) `waitForMessage()` method. This method will return a handle to the worker that sent a message. Alternatively, the manager can invoke `getNextMessage()` (again, blocking optional) to immediately receive the next message.

Listing 8.5 and Listing 8.6 are continuations of Listing 8.3 on page 25 and Listing 8.4 on the previous page respectively, showing an example of communication, using `std::string` messages.

Listing 8.5: `Worker` Communication

```
        Memory::uint8Array msg;

        /* Deal with next customer unless Manager interrupts in next second */
        if (this->waitForMessage(1)) {
                if (this->receiveMessageFromManager(msg)) {
                        Action action = Restaurant::messageToAction(msg);
                        switch (action) {
                        case TAKE_BREAK:
                                employee->goOnBreak();
                                break;
                        /* ... */
                        }
                }
        }

        /* ... */

        if (customer->isComplaining()) {
                sprintf((char *)&(*msg), "Customer Complant");
                this->sendMessageToManager(msg);
        }
```

Listing 8.6: `Manager` Communication

```
        tr1::shared_ptr<Process::WorkerController> sender;
        Memory::uint8Array msg;

        /* Do routine tasks unless employee has concern in the next 2 seconds */
        while (this->getNextMessage(sender, msg, 2)) {
                Action action = Restaurant::messageToAction(msg);
                switch (action) {
```

```
            case CUSTOMER_COMPLAINT:
                sprintf((char *)&(*msg), "I'll take care of it.");
                this->sendMessageToWorker(msg);
                break;
            /* ... */
            }
    }

    /* ... */

    /*  Closing Time */
    sprintf((char *)&(*msg), "Clock out and go home.");
    this->broadcastMessage(msg);
```

Chapter 9

System

The `System` package provides a set of functions in the that return information about the hardware and operating system. This information can be used by applications to determine the amount of real memory, number of central processing units, or current load average. This information can be used to dynamically tailor the application behavior, or simply to provide additional information in a runtime log.

Listing 9.1 shows how an application can spawn several child processes based on the number of CPUs and memory available. Note that this information may not be available on all platforms, and therefore, the application must be prepared to handle that situation.

Listing 9.1: Using the `System` CPU Count Information

```
#include <iostream>
#include <be_system.h>

using namespace BiometricEvaluation;

int
main(int argc, char* argv[]) {

    // perform some application setup ...

    uint32_t cpuCount;
    uint64_t memSize, vmSize;
    try {
        cpuCount =  System::getCPUCount();
        cpuCount--;    // subtract one CPU for the parent process
        memSize = System::getRealMemorySize();
        Process::Statistics::getMemorySizes(NULL, &vmSize, NULL, NULL, NULL);
        memSize -= vmSize;   // subtract off memory used by parent

        // Give each child a fraction of the memory
        spawnChildren(cpuCount, memSize / cpuCount);
    } catch (Error::NotImplemented) {
            cout << "Running a single process only." << endl;
    }

    // processing done by parent ...
}
```

Chapter 10

Image

The `Image` package maintains the classes and other information related to images and image processing. Within the Biometric Evaluation Framework, many classes refer to images, such as when dealing with fingerprint data. Many biometric data standards supply the actual image encoded in one of several standard formats. Applications can retrieve the image as stored in the record, or decompressed by the Image class into a “raw” format. Therefore, within the BECommon, several of the common compression formats are supported, removing the need for applications to decompress the image directly, while maintaining access to the as-recorded image format.

10.1 The Image Namespace

The `Image` namespace contains several data types used to represent aspects of an image. The types defined are chiefly used to retrieve common information from images stored in an `Image` class (section 10.2). Data types in the Image namespace do not perform any translation of scale units or sizing, as each set of attributes is copied directly from the image data itself when possible.

The same applies to images encapsulated in biometric records. Although some biometic records have fields for image attributes like dimensions and resolution, the corresponding fields of an `Image` class are **not** populated with their contents. The `Image` namespace data types *are* used outside of the namespace, such as in finger views, to retrieve image attributes stored as part of the biometric record. Applications can compare those values against the values within the `Image` object, as in most cases those values are taken directly from the underlying image data. See Chapter 14 on page 45 for more information on image-based biometric records.

The `Image` namespace contains all of the `Image` classes that are used to represent an image. These classes are described in the following sections.

10.2 The Image Class

The `Image` class is an abstract base class that defines a set of minimum functionality for all supported image formats. Once an `Image` has been constructed, it may not be modified. For any supported image format, the following information is required to be accessible:

- Original binary data
- Compression algorithm
- Decompressed (“raw”) format binary data (grayscale, full color)
- Depth

- Dimensions (width, height)
- Resolution (horizontal, vertical)

A rudimentary implementation of generating a grayscale image is provided by the `Image` class in `getRawGrayscaleData()`. This implementation calculates the luminance value Y (of YCbCr) for each pixel of a color image. The resulting image always uses 8-bits to represent a pixel, but can return a raw image using 2 gray levels (1-bit) or 256 gray levels (8-bit). The 1-bit algorithm quantizes to black when the 8-bit color value is $\leq$127. `Image` subclasses may override and implement their own grayscale conversion methods.

Also of interest in the Image class is `valueInColorspace()`, a static function to convert color values between bit depths.

10.3 Raw Image

The `RawImage` class represents a decompressed image, or an image where `getRawData()` would return the exact same data as `getData()`. `RawImage` has no special implementation or additional methods.

10.4 JPEG

The `JPEG` class represents an image encoded according to the JPEG image standard [8]. Decompression and grayscale conversion are accomplished via `libjpeg` [6].

As of version 8.0, `libjpeg` provided a way to handle JPEG images existing within in-memory buffers, as opposed to on-disk files. Because the Image class requires in-memory buffers, `JPEG` includes a JPEG memory source manager implementation, but it is built only if a version of `libjpeg` older than 8.0 is detected at compile-time.

`JPEG` provides a static function to determine whether or not a data buffer appears to be encoded in the JPEG image standard format. Errors within libjpeg will be caught and rethrown as `Exception` s.

10.5 JPEGL

Similar to `JPEG`, the `JPEGL` class performs `Image` class services for lossless JPEG encoded images. JPEGL decompression is performed by NIST Biometric Image Software 's `libjpegl` [13].

10.6 JPEG2000

The `JPEG2000` class provides Image class functionality to JPEG 2000-encoded images [7]. The class makes an attempt to support the following JPEG 2000 codecs:

- JPEG 2000 codestream (.j2k)
- JPEG 2000 compressed image data (.jp2)
- JPEG 2000 interactive protocol (.jpt)

Decompression is provided by the OpenJPEG library (`libopenjpeg`) [11]. `JPEG2000` also provides a static function to test whether or not an image appears to be JPEG 2000-encoded.

Not all information required by the `Image` class is present in a JPEG 2000-encoded image. In particular, some codecs and encoders omit the “Display Resolution Box.” It is generally accepted that the resolution will be 72 pixels-per-inch when the “Display Resolution Box” is not present.

Errors within `libopenjpeg` will be caught and rethrown as `Exception` s.

10.7 NetPBM

The `NetPBM` class provides Image class functionality to all types of NetPBM formatted images, up to 48-bit depth. This includes the following formats:

- ASCII Portable Bitmap (P1, .pbm)
- ASCII Portable Graymap (P2, .pgm)
- ASCII Portable Pixmap (P3, .ppm)
- Binary Portable Bitmap (P4, .pbm)
- Binary Portable Graymap (P5, .pgm)
- Binary Portable Pixmap (P6, .ppm)

`NetPBM` provides some of its more general use parsing algorithms as static functions for use outside of the class. This includes ASCII to binary pixel conversion. A function to test for NetPBM formats is also provided.

10.8 PNG

The `PNG` class represents an image encoded according to the PNG image standard [5]. Decompression is provided by `libpng` [15].

`PNG` provides a static function to test whether or not an image appears to be encoded in the PNG image standard format. Errors within `libpng` are caught and rethrown as `Exception` s.

10.9 WSQ

Images encoded in the WSQ-image standard [18] are represented by the `WSQ` class. The WSQ decompressor found in NIST Biometric Image Software [13], `libwsq`, is used by this class. The class provides a static function to determine whether or not an image appears to be encoded in the WSQ format.

Errors from the `libwsq` will be displayed through `stderr` and will **not** be rethrown as `Exception` s.

Chapter 11

Text

The `Text` package consists of functions to perform common operations on `string`s and `char` arrays. Many of the operations may be considered "trivial," but are used often enough within the Biometric Evaluation Framework and other applications that a common implementation in BECommon is more than warranted. A complete listing of functions is available in the documentation appendix for `BiometricEvaluation::Text`2.

Listing 11.1 shows how to use the `split()` function from the `Text` package. `split()` can separate a `string` into tokens delimited by a character, useful for processing comma- or space-separated text files (such files could be produced by a `LogSheet` (Section 6.3 on page 16), for instance). Here, a text file containing metadata for an image is being parsed, perhaps to be passed to the `RawImage` constructor (Section 10.3 on page 34).

Listing 11.1: Tokenizing a `string`

```
/* Definition of input strings */
static const vector<string>::size_type filenameToken = 0;
static const vector<string>::size_type widthToken = 1;
static const vector<string>::size_type heightToken = 2;
static const vector<string>::size_type depthToken = 3;

/* Split the string, presumably input from a file */
string input = "/mnt/raw\\ images/1.raw 500 500 8";
vector<string> tokens = Text::split(input, ' ', true);

/* Assign the retrieved tokens */
string filename;
uint32_t width, height, depth;
try {
        filename = tokens.at(filenameToken);    /* "/mnt/raw images/1.raw" */
        width = atoi(tokens.at(widthToken).c_str());    /* "500" */
        height = atoi(tokens.at(heightToken).c_str());  /* "500" */
        depth = atoi(tokens.at(depthToken).c_str());    /* "8" */
} catch (out_of_range) {
        throw Error::FileError("Malformed input");
}
```

Notice the `true` parameter to `split()` in Listing 11.1. This instructs `split()` to not tokenize based on an escaped delimiter. If `false`, the first token would be split into two at the presence of the delimiter.

`Text` also contains functions to perform hashing via `OpenSSL`. A two-line program that emulates the command-line `md5sum` program is shown in Listing 11.2. Changing the digest parameter to `"sha1"` would make the program emulate `` `openssl sha1` ``.

Listing 11.2: `md5sum` via BECommon

```
#include <cstdlib>
#include <iostream>

#include <be_io_utility.h>
#include <be_text.h>
#include <be_memory_autoarray.h>

using namespace std;
using namespace BiometricEvaluation;

int
main(
    int argc,
    char *argv[])
{
        if (argc == 0)
                return (EXIT_FAILURE);

        try {
                Memory::uint8Array file = IO::Utility::readFile(argv[1]);
                cout << Text::digest(file, file.size(), "md5") << "  " <<
                    argv[1] << endl;
        } catch (Error::Exception) {
                return (EXIT_FAILURE);
        }

        return (EXIT_SUCCESS);
}
```

Chapter 12

Feature

The Feature package contains those items that relate to the representation of biometric features, such as fingerprint minutiae, facial features (eyes, etc.), and related information. Objects of these class types are typically associated with View (Chapter 14 on page 45) or DataInterchange (Chapter 15 on page 47) objects. For example, a minutiae object is usually obtained from a finger view, which may have been obtained from a data interchange object representing an entire biometric record for an individual.

The data contained within a Feature object is represented as the “native” format as it was extracted from the underlying data record. There is no translation to a common format and it is the application’s responsibility to interpret or translate the data as necessary.

Currently, fingerprint and palm print minutiae are the features supported within the BECommon. As development continues, additional features contained within biometric data records will be supported.

12.1 ANSI/NIST Features

The ANSI/NIST [3] standard defines several features represented as data elements within a record. Fingerprint and palm minutiae is contained within Type-9 record. The AN2K7Minutiae class, contained in the Feature package, represents a single Type-9 record. An object of this class can be constructed directly from a complete ANSI/NIST record. However, it is more common for an application to retrieve these objects from the AN2KView object defined in the Finger package (Chapter 13 on page 41).

See Listing 13.1 on page 42 for a complete example of how to obtain the fingerprint minutiae data from an ANSI/NIST record.

12.2 ISO/INCITS Features

The ISO [2] and INCITS [1] fingerprint minutiae standards are represented within BECommon with the same class, INCITSMinutiae, as the minutiae format is identical in both standards.

Listing 13.2 on page 43 shows how to create a view object for the fingerprint minutiae record contained in a file.

Chapter 13

Finger

One of the most commonly used biometric source is the fingerprint. Multiple types of information can be derived from a fingerprint, including minutiae and the pattern, such as whorl, etc. The `Finger` package contains the types, classes, and other items that are related to fingers and fingerprints. Objects of the `Finger` classes are typically not used in a stand-alone fashion, but are usually obtained from an object in the `DataInterchage` (Chapter 15 on page 47) package.

Several enumerated types are defined in the `Finger` package. The types are used to represent those elements related to fingers and fingerprints that are common across all data formats. Types that represent finger position, impression type, and others are included in the package. Stream operators are defined for these types so they can be printed in human-readable format.

Most of the classes in the `Finger` package represent data taken directly from a record in a standard format (e.g. ANSI/NIST [3]). In addition to general information, such as finger position, other information may be represented: The source of the finger image; the quality of the image, etc. In addition to this descriptive information, the finger object will provide the set of derived minutiae or other data sets.

When representing the information about a finger (and fingerprint), the class in the `Finger` package implements the interface defined in the `View` package. A finger is a specific type of view in that it represents all the available information about the finger, including the source image, minutiae (often in several formats), as well as the capture data (date, location, etc.)

13.1 ANSI/NIST Minutiae Data Record

Finger views are objects that represent all the available information for a specific finger as contained in one or more biometric records. For example, an ANSI/NIST file may contain a Type-3 record (finger image) and an associated Type-9 record (finger minutiae). A finger view object based on the ANSI/NIST record can be instantiated and used by an application to retrieve all the desired information, including the source finger image. The internals of record processing and error handling are encapsulated within the class.

The BECommon provides several classes that are derived from a base View class, contained within the `Finger` package. See Chapter 13 for more information on the types associated with fingers and fingerprints. This section discusses finger views, the classes which are derived from the general `View` class. These subclasses represent specific biometric file types, such as ANSI/NIST or INCITS/M1. In the latter case, two files must be provided when constructing the object because INCITS finger image and finger minutiae records are defined in two separate standards.

13.1.1 ANSI/NIST Finger Views

An ANSI/NIST record may contain one or more finger views, each based on a type of finger image. These Type-3, Type-4, etc. records contain the image and Type-9 minutiae data, among other information. These record types are grouped into either the fixed- or variable-resolution categories, and are represented as specific classes within BECommon, `AN2KViewFixedResolution` and `AN2KViewVariableResolution`.

The `AN2KMinutiaeDataRecord` class represents all of the information taken from a ANSI/NIST Type-9 record. A Type-9 record may include minutiae data items in several formats (standard and proprietary) and the impression type code.

Listing 13.1 shows how an application can use the `AN2KViewFixedResolution` to retrieve image information, image data, and derived minutiae information from a file containing an ANSI/NIST record with Type-3 (fixed resolution image) and Type-9 (fingerprint minutiae) records.

Listing 13.1: Using an AN2K Finger View

```
#include <fstream>
#include <iostream>
#include <be_finger_an2kview_fixedres.h>
using namespace std;
using namespace BiometricEvaluation;

int
main(int argc, char* argv[]) {

    Finger::AN2KViewFixedResolution *_an2kv
    try {
        _an2kv = new Finger::AN2KViewFixedResolution("type9-3.an2k",
            TYPE_3_ID, 1);
    } catch (Error::DataError &e) {
        cerr << "Caught " << e.getInfo()  << endl;
        return (EXIT_FAILURE);
    } catch (Error::FileError& e) {
        cerr << "A file error occurred: " << e.getInfo() << endl;
        return (EXIT_FAILURE);
    }
    std::auto_ptr<Finger::AN2KView> an2kv(_an2kv);

    cout << "Image resolution is " << an2kv->getImageResolution() << endl;
    cout << "Image size is " << an2kv->getImageSize() << endl;
    cout << "Image depth is " << an2kv->getImageDepth() << endl;
    cout << "Compression is " << an2kv->getCompressionAlgorithm() << endl;
    cout << "Scan resolution is " << an2kv->getScanResolution() << endl;

    // Save the finger image to a file.
    tr1::shared_ptr<Image::Image> img = an2kv->getImage();
    if (img.get() == NULL) {
       cerr << "Image was not present." << endl;
       return (EXIT_FAILURE);
    }
    string filename = "rawimg";
    ofstream img_out(filename.c_str(), ofstream::binary);
    img_out.write((char *)&(img->getRawData()[0]),
        img->getRawData().size());
    if (img_out.good())
            cout << "\tFile: " << filename << endl;
    else {
        img_out.close();
        cerr << "Error occurred when writing " << filename << endl;
        return (EXIT_FAILURE);
    }
```

```
    img_out.close();

    // Get the finger minutiae sets. AN2K records can have more than one
    // set of minutiae for a finger.

    vector<Finger::AN2KMinutiaeDataRecord> mindata = an2kv->getMinutiaeDataRecordSet();
}
```

13.1.2 ISO/INCITS Finger Views

The ISO [10] and INCITS [9] standards typically use separate files for the source biometric data and the derived data. For example, the ISO 19794-2 standard is for fingerprint minutiae data, while 19794-4 is for finger image data. The corresponding BECommon view objects are constructed with both files, although a view can be constructed with only one file. In the latter case, the view object will represent only that information contained in the single file.

Listing 13.1 on the preceding page shows how an application can create a view from a ANSI/INCTIS 378 finger minutiae format record [1].

Listing 13.2: Using an INCITS Finger View

```
#include <stdlib.h>
#include <fstream>
#include <iostream>
#include <be_finger_ansi2004view.h>
#include <be_feature_incitsminutiae.h>
using namespace std;
using namespace BiometricEvaluation;

int
main(int argc, char* argv[]) {

    Finger::ANSI2004View fngv;
    try {
        fngv = Finger::ANSI2004View("test_data/fmr.ansi2004", "", 3);
    } catch (Error::DataError &e) {
        cerr << "Caught " << e.getInfo()  << endl;
        return (EXIT_FAILURE);
    } catch (Error::FileError& e) {
         cerr << "A file error occurred: " << e.getInfo() << endl;
         return (EXIT_FAILURE);
    }
    cout << "Image resolution is " << fngv.getImageResolution() << endl;
    cout << "Image size is " << fngv.getImageSize() << endl;
    cout << "Image depth is " << fngv.getImageDepth() << endl;
    cout << "Compression is " << fngv.getCompressionAlgorithm() << endl;
    cout << "Scan resolution is " << fngv.getScanResolution() << endl;

    Feature::INCITSMinutiae fmd = fngv.getMinutiaeData();
    cout << "Minutiae format is " << fmd.getFormat() << endl;
    Feature::MinutiaPointSet mps = fmd.getMinutiaPoints();
    cout << "There are " << mps.size() << " minutiae points:" << endl;
    for (size_t i = 0; i < mps.size(); i++)
        cout << mps[i];

        Feature::RidgeCountItemSet rcs = fmd.getRidgeCountItems();
```

```
    cout << "There are " << rcs.size() << " ridge count items:" << endl;
    for (int i = 0; i < rcs.size(); i++)
        cout << "\t" << rcs[i];

    Feature::CorePointSet cores = fmd.getCores();
    cout << "There are " << cores.size() << " cores:" << endl;
    for (int i = 0; i < cores.size(); i++)
        cout << "\t" << cores[i];

    Feature::DeltaPointSet deltas = fmd.getDeltas();
    cout << "There are " << deltas.size() << " deltas:" << endl;
    for (int i = 0; i < deltas.size(); i++)
        cout << "\t" << deltas[i];

    exit (EXIT_SUCCESS);
}
```

Chapter 14

View

Within the Biometric Evaluation Framework a `View` represents all the information that was derived from an image of a biometric sample. For example, with a fingerprint image, any minutiae that were extracted from that image, as well as the image itself, are contained within a single `View` object. In many cases the image may not be present, however the image size and other information is contained within a biometric record, along with the derived information. A `View` is used to represent these records as well.

In the case where a raw image is part of the biometric record, the `View` object's related `Image` (Chapter 10 on page 33 object will have identical size, resolution, etc. values because the `View` class sets the `Image` attributes directly. For other image types (e.g. `JPEG`) the `Image` object will return attribute values taken from the image data.

`View` s are high-level abstractions of the biometric sample, and concrete implementations of a `View` include finger, face, iris, etc. views based on a specific type of biometric. Therefore, `View` objects are not created directly, Subclasses, such as finger views (see Chapter 13 on page 41), represent the specific type of biometric sample.

Objects are created with information taken from a biometric data record, an ANSI/NIST 2007 file, for example. Most record formats contain information about the image itself, such as the resolution and size. The `View` object can be used to retrieve this information. However, the data may differ from that contained in the image itself, and applications can compare the corresponding values taken from the `Image` object (when available) to those taken from the `View` object.

Chapter 15

Data Interchange

The DataInterchange package consists of classes and other elements used to process an entire biometric data record, or set of records. For example, a single ANSI/NIST record, consisting of many smaller records (fingerprint images, latent data, etc.) can be accessed by instantiating a single object. Classes in this package typically use has-a relationships to classes in the Finger and other packages that process individual biometric samples.

The design of classes in the DataInterchange package allows applications to create a single object from a biometric record, such as an ANSI/NIST file. After creating this object, the application can retrieve the needed information (such as Finger View s Chapter 13 on page 41) from this object. A typical example would be to retrieve all images from the record and pass them into a function that extracts a biometric template or some other image processing.

15.1 ANSI/NIST Data Records

The ANSI/NIST Data Interchange package contains the classes used to represent ANSI/NIST [3] records. One class, AN2KRecord, is used to represent the entire ANSI/NIST record. An object of this class will contain objects of the Finger classes, as well as other packages. By instantiating the AN2KRecord object, the application can retrieve all the information and images contained in the ANSI/NIST record.

The AN2KMinutiaeDataRecord class represents an entire Type-9 record from an ANSI/NIST file. However, some components of this class are represented by classes in other packages. For example, the AN2K7Minutiae class in the Feature package represents the "standard" format minutiae in the Type-9 record.

Listing 15.1 shows how an application can retrieve all finger captures (Type-4 records) from an ANSI/NIST record. Once the Views are retrieved, the application obtains the set of minutiae records associated with that View.

Listing 15.1: Retrieving ANSI/NIST Finger Captures

```
#include <iostream>
#include <be_error_exception.h>
#include <be_finger_an2kview_capture.h>

int
main(int argc, char* argv[])
{
    /*
     * Call the constructor that will open an existing AN2K file and
     * retrieve the first finger capture (Type-14) record.
     */
```

```
    std::auto_ptr<Finger::AN2KViewCapture> an2kv;
    try {
        an2kv.reset(new Finger::AN2KViewCapture("type9-14.an2k", 1));
    } catch (Error::DataError &e) {
        cout << "Caught " << e.getInfo()  << endl;
        return (EXIT_FAILURE);
    } catch (Error::FileError& e) {
        cout << "A file error occurred: " << e.getInfo() << endl;
        return (EXIT_FAILURE);
    }

    cout << "Get the set of minutiae data records: ";
    vector<Finger::AN2KMinutiaeDataRecord> records =
        an2kv->getMinutiaeDataRecordSet();
    cout << "There are " << records.size() << " minutiae records." << endl;

    /*
     * Get the info from the first minutiae record in the View.
     */
    DataInterchange::AN2KMinutiaeDataRecord type9 = records[0];

    /*
     * Get the "standard" set of minutiae.
     */
    Feature::AN2K7Minutiae an2k7m = type9.getAN2K7Minutiae();

    /*
     * Obtain the minutiae points, ridge counts, cores, and deltas.
     */
    Feature::MinutiaPointSet mps;
    Feature::RidgeCountItemSet rcs;
    Feature::CorePointSet cps;
    Feature::DeltaPointSet dps;
    try {
        mps = an2k7m->getMinutiaPoints();
        rcs = an2k7m->getRidgeCountItems();
        cps = an2k7m->getCores();
        dps = an2k7m->getDeltas();

    } catch (Error::DataError &e) {
        cout << "Caught " << e.getInfo()  << endl;
        return (EXIT_FAILURE);
    }

    cout << "There are " << mps.size() << " minutiae points:" << endl;
    /*
     * Print out the minutiae points.
     */
    for (int i = 0; i < mps.size(); i++) {
        printf("(%u,%u,%u)\n", mps[i].coordinate.x, mps[i].coordinate.y,
             mps[i].theta);
    }
    cout << "There are " << rcs.size() << " ridge counts:" << endl;
    for (int i = 0; i < rcs.size(); i++) {
        printf("(%u,%u,%u)\n", rcs[i].index_one, rcs[i].index_two,
        rcs[i].count);
```

```
    }
    cout << "There are " << cps.size() << " cores." << endl;
    cout << "There are " << dps.size() << " deltas." << endl;

    cout << "Fingerprint Reader: " << endl;
    try { cout << an2k7m->getOriginatingFingerprintReadingSystem() << endl; }
    catch (Error::ObjectDoesNotExist) { cout << "<Omitted>" << endl; }

    cout << "Pattern (primary): " <<
    Feature::AN2K7Minutiae::convertPatternClassification(
    an2k7m->getPatternClassificationSet().at(0)) << endl;

    return(EXIT_SUCCESS);
}
```

Listing 15.2 shows how an application can retrieve all latent finger images from a set of ANSI/NIST record retrieved from a `RecordStore`. Using the `Image` object, the image's "raw" data can be retrieved and passed to another function for processing. Note that the image data may be stored in a compressed format inside the ANSI/NIST record, but is converted to raw format by the `Image` object.

Listing 15.2: Retrieving ANSI/NIST Latent Records

```
#include <be_io_recordstore.h>
#include <be_data_interchange_an2k.h>
using namespace BiometricEvaluation;

void
processImageData(uint8_t *buf, uint32_t size)
{
    :
    :
    :
    :
}

int
main(int argc, char* argv[]) {

    std::tr1::shared_ptr<IO::RecordStore> rs;
    try {
        rs = IO::RecordStore::openRecordStore(rsname, datadir, IO::READONLY);
    } catch (Error::Exception &e) {
        cerr << "Could not open record store: " << e.getInfo() << endl;
        return (EXIT_FAILURE);
    }

    /*
     * Read some AN2K records and construct the View objects.
     */
    Utility::uint8Array data;
    string key;
    while (true) {          // Loop through all records in store
        uint64_t rlen;
        try {
            rlen = rs->sequence(key, NULL);
        } catch (Error::ObjectDoesNotExist &e) {
            break;
```

```
        } catch (Error::Exception &e) {
            cout << "Failed sequence: " << e.getInfo() << endl;
            return (EXIT_FAILURE);
        }
        data.resize(rlen);
        try {
            rs->read(key, data);
            DataInterchange::AN2KRecord an2k(data);
            std::vector<Finger::AN2KViewLatent> latents = an2k.getFingerLatents();
            for (int i = 0; i < latents.size(); i++) {
                tr1::shared_ptr<Image::Image> img = latents[i].getImage();
                if (img != NULL) {
                    cout << "\tCompression: " << img->getCompressionAlgorithm() << endl;
                    cout << "\tDimensions: " << img->getDimensions() << endl;
                    cout << "\tResolution: " << img->getResolution() << endl;
                    cout << "\tDepth: " << img->getDepth() << endl;
                    processImageData(img->getRawData(), img->getRawData().size());
                }
            }
        } catch (Error::Exception &e) {
            return (EXIT_FAILURE);
        }
    }
    return(EXIT_SUCCESS);
}
```

15.2 INCITS Data Records

This INCITS class of data records covers all those record formats that are derived from the standards defined by the InterNational Committee for Information Technology Standards [9]. These formats include the ANSI-2004 Finger Minutiae Record Format [1], the ISO equivalent [2], and other data formats, including finger images.

15.2.1 Finger Views

Within the BECommon, finger view objects (Section 14) can be created from a combination of finger minutiae and image records. However, it is not necessary to have both records in order to create the view because each record contains enough information to represent the finger (image size, for example). However, if a view is contructed using only the minutiae record, then the image itself will not be present. Alternatively, if a view is made from an image record, no minutiae data would be available. It is possible to construct a view without any information.

Listing 13.2 on page 43 shows an example of accessing the information in an ANSI 378-2004 Finger Minutiae Record by creating an `ANSI2004View` object from the record file.

Chapter 16

Messaging

Biometric Evaluation Framework contains a collection of classes to facilitate reciving messages asynchronously over a network. What is done with these messages and how (or if) to respond is ultimately up to the application. BECommon uses this messaging in a concrete way to receive text-based commands from a `telnet` session over the Internet.

16.1 Message Center

`Process::MessageCenter` is the public-facing class an application uses to receive messages over a network. A *message* is a user-defined blob of data stored in an array of bytes. Instantiate a `MessageCenter`, and it will dilligently await connections on the specified port in a separate process. During its run-loop, the appplication may poll or wait to determine if a message is waiting. The application has the choice of dealing with the message, sending a response, or ignoring the message entirely. Because the `MessageCenterListener` is in a separate process, the main run-loop of the application does not have to be interrupted. The `MessageCenter` classes utilize existing framework inter-process communication techniques to propagate messages (see Subsection 8.2.4 on page 28).

Listing 16.1: Basic `MessageCenter` Usage

```
namespace BE = BiometricEvaluation;

uint32_t clientID;
BE::Memory::uint8Array message;
BE::Process::MessageCenter mc;
for (;;) {
        /* ... do work ... */

        if (mc.hasUnseenMessages()) {
                mc.getNextMessage(clientID, message);
                std::cout << clientID << " sent a " << message.size() <<
                    " byte message." << std::endl;

                Memory::AutoArrayUtility::setString(message, "ACK\n");
                mc.sendResponse(clientID, message);
        }
}
```

Messages can be sent to the `MessageCenter` in a number of ways, like `telnet` connections or `write()` ing to a socket. Messages are terminated with a newline (`\n`) character.

16.2 Command Center

It's easy to see how `MessageCenter` might be used for passing *commands* to a running application. One might want to query the *status* of an operation or ask a process to *stop*. The aim of `CommandCenter` was to take this common command-passing pattern and make it easier.

With `CommandCenter`, an application defines one or more `enum class` es using `Framework::Enumeration` s (see Section 3.2 on page 5). For convenience, the application should subclass the `CommandParser template`, with the enumeration as the templated type. The base class instantiates a `MessageCenter` and listens for connections. Just like `MessageCenter`, commands do not have to be dealt with or responded to, and the application will only know if a command is awaiting a response if the application asks.

Because `CommandParser` operates off of strongly-typed enumerations, a pure virtual method, `parse(Command)`, must be implemented in the child class. It is expected that this method will simply be a `switch` statement of all possible enumerations (*commands*). The body of the `switch` will likely call other methods, each dealing with a single command.

`CommandParser` performs some additional convenience functions to help application developers quickly respond to commands. A *usage* string may be automatically sent when an invalid command is received. The application's main run-loop will never see the failed command attempt. If a valid command is received, `CommandParser` will tokenize any extra text in the sent command and store it in an easily retrieved `vector`. The method called from `parse()` can then sanity-check the arguments and send an error message back to the client if the arguments are invalid.

Listing 16.2: Basic `CommandCenter` Usage

```
namespace BE = BiometricEvaluation;

enum class TestCommand
{
        Stop,
        Help
};

template<>
const std::map<TestCommand, std::string>
BE::Framework::EnumerationFunctions<TestCommand>::enumToStringMap {
        {TestCommand::Stop, "STOP"},
        {TestCOmmand::Help, "HELP"}
};

class TestCommandParser : public BE::Process::CommandParser<TestCommand>
{
public:
        void
        parse(
            const BE::Process::CommandParser<TestCommand>::Command &command)
        {
                switch (command.command) {
                case TestCommand::Stop:
                        this->stop(command);
                        break;
                case TestCommand::Help:
                        this->help(command);
                        break;
                }
        }

```

```
private:
        void
        stop(
            const BE::Process::CommandParser<TestCommand>::Command &command)
        {
                /* Ensure proper arguments */
                if (command.arguments.size() != 1) {
                        this->sendResponse(command.clientID, "Usage: " +
                            to_string(command.command) + " <process>");
                        return;
                }

                /* ... perform stop operation ... */
        }

        void
        help(
            const BE::Process::CommandParser<TestCommand>::Command &command)
        {
                this->sendResponse(command.clientID, "Available Commands:\n"
                    "\tSTOP <process>\n\tHELP");
        }
};

int
main()
{
        TestCommandParser commandCenter;
        TestCommandParser::Command command;
        for (;;) {
                /* ... do work ... */

                if (commandCenter.hasPendingCommands()) {
                        commandCenter.getNextCommand(command);
                        commandCenter.parse();
                }
        }

        return (EXIT_SUCCESS);
}
```

It's perfectly acceptible for an application to make use of more than one `CommandParser` for different `enum` s, assuming they are listening on different ports.

Chapter 17

Parallel Processing

17.1 MPI Parallel Processing Package

The `MPI` package is a set of APIs used implement parallel processing using the MPI [12] network-based messaging system. The core concept implemented in the framework is that of a distributor, one or more receivers, work packages, and a processing element to be implemented by the application.

The classes that make up the MPI package encapsulate all the necessary function calls and error handling in order to create an MPI job. Furthermore, the distribution and reception of packages containing data to be used for processing are also encapsulated within the MPI Framework. Lastly, logging, both for the tracing of Framework activity as well as application needs, is managed by these classes.

Figure 17.1 on the following page shows the processes and data flow for a typical parallel job using components of the Framework. The distributor process executes code from the `Distributor` class, and the receiver process likewise executes `Receiver` class code. Within each process is shown the Framework packages that could be used for the job. The *Lib* element refers to the "black-box" component of software being tested, a fingerprint matching library, for example. In this example, a record store is used as the data source, and record keys are sent in the work packages. On the receiving side, the keys are used to read record data (values) from the same store.

On the receiving side of the job, the processing is separated into two areas of responsibility. Each Task-N is responsible for managing the workers (Task-N:1 . . . Task-N:c) by starting them, accepting work requests, and sending a command to have them shut down when the job finishes. Each worker is responsible for consuming the contents of the work packages; that implementation is done in the application.

The partitioning of responsibility enables two features of the Framework. First, a worker process can handle signals or other errors and decide to shutdown without affecting the rest of the job. This capability is important when testing "black-box" software where function calls cannot be trusted.

Second, each Task-N can perform some work before creating the workers. One example is the loading of a large data set into memory; again, this is done within the application. Because Task-N calls the POSIX function `fork()` to create the workers, each worker inherits the work done by Task-N. In the case of a memory load, each worker now has that memory mapped into it's address space. See Section 17.5 on page 57 for more details.

17.2 Work Package

A `WorkPackage` object wraps a simple container of data with some access methods. There is no information in this class pertaining to the nature or format of the data; it is simply treated as an array of unsigned integer values. However, clients of the class can store a value, the "number of elements", that is transmitted along with the package. This value only has meaning to the client, and is usually equivalent to the number of larger-sized components making up the package. For example, this value may be the number of records contained in the package. It is up to the client of `WorkPackage` to understand how to separate the array into components.

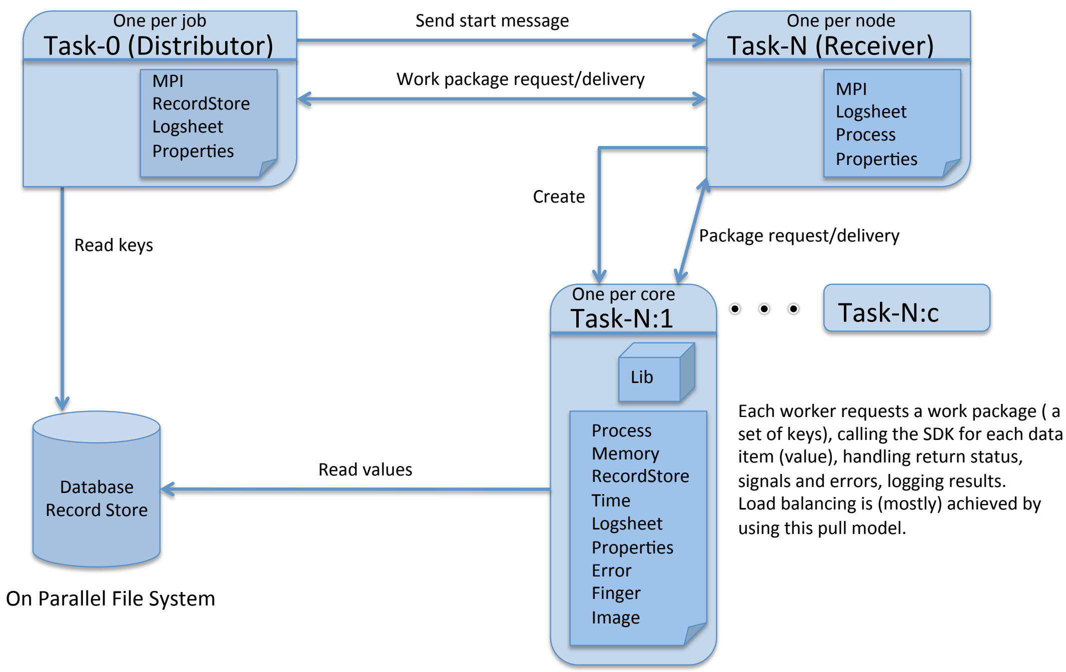

Figure 17.1: MPI Parallel Job Processes and Data Flow

The classes `RecordStoreDistributor` (Section 17.3.1) and `RecordProcessor` (Section 17.5.1 on the next page) are examples of `WorkPackage` clients that insert and remove data from a work package.

17.3 Distributor

The `Distributor` is an abstract class than encapsulates the MPI functionality and is responsible for distributing work packages to other elements within the MPI job (the receivers). However, this class is also responsible for coordinating the startup and shutdown of the receiver tasks. MPI messages are used for this coordination. An MPI job may fail to start if the distributor fails to initialize, or if none of the receivers initialize.

One method of the `Distributor` class, `createWorkPackage()`, is implemented by child classes. This method creates a single work package with the knowledge of how the elements of the package are to be stored in the package's data buffer. `RecordStoreDistributor` is an implementation of `Distributor`.

17.3.1 Record Store Distributor

`RecordStoreDistributor` reads records from a `RecordStore`, packs record keys, and optionally, values into a `WorkPackage`. This class inherits all of the MPI communication, intra-job coordination, logging, and other aspects of the `Distributor` parent class.

An application can create an instance of a `RecordStoreDistributor` with the name of a record store in order to distribute records for processing across the MPI job. Listing 17.3 on page 63 shows an example section of code to create a record store distributor. In this type of application there is no need for the application code to refine any of the Framework classes.

17.4 Receiver

The `Receiver` class encapsulates all the MPI messaging needed to participate in the MPI job as the receiver of data to be processed. In addition, this class is responsible for starting other processes that perform work on the actual data from the work package.

It is expected, as part of the MPI job, that a single receiver process will be started on each node in the job. More than one can be started, however. Each receiver starts one or more child processes to consume data. The receiver monitors each worker process and will instruct them to shut down when the job is finished (no more data), early termination signals are received, or in the case of errors encountered by the receiver.

By keeping the data consumers as separate processes, the receiving half of the MPI job can be more robust as a premature termination of a worker process (due to memory corruption, for example) will not affect other workers.

17.5 Work Package Processor

The `WorkPackageProcessor` class is pure-virtual and provides the interface for any class that uses a `WorkPackage` to receive data from the MPI Framework. `WorkPackageProcessor` also maintains a `Logsheet` object which can be used by subclasses to store log messages.

Implementations of this class can be considered to have dual responsibilities. First is the management of common state used by all workers (Task-N:c in Figure 17.1 on the preceding page); creating state data shared by all workers, for example. Second, as a factory to create a package consumer for the worker process.

The `performInitialization()` method is called before the `Receiver` object forks and creates the worker processes. The application can use this function to load a large data set into memory (taking advantage of copy-on-write memory semantics present in most modern operating systems), or perform any node-local setup that should only be done once the MPI job has begun.

`newProcessor()` returns a new instance of the package processor. This method is called by the Framework when a new process is started by the receiver to consume work packages sent by the distributor. This method is a factory, creating new instances of the `WorkPackageProcessor` implementation. Therefore, it must create a "fully-formed" object that may have different state than that created by the class constructor. An example would be creating an output log file with record information. This output file would not be created in the constructor because the object returned from that will not process a work package; it is the factory object.

It is the responsibility of the `newProcessor()` method to ensure there is no resource contention between instances of this class, as the methods of this object will be executed within a separate process. The `MPI::generateUniqueID()` function can be used to create a name string that to identify the process.

17.5.1 Record Processor

`RecordProcessor` is a partial implementation of `WorkPackageProcessor` and defines the `processWorkPackage()` of the `WorkPackageProcessor` interface; other methods are declared as pure-virtual and must be implemented by a child class. In addition, `RecordProcessor` declares a new pure-virtual method, `processRecord()` to be implemented by a subclass to process a single record from the record store. In summary, `RecordProcessor` removes records from the work package to be processed within the subclass, which is defined by the application. See Listing 17.1 on the facing page and Listing 17.2 on page 60 for a example of such an implementation.

17.6 MPI Resources

Every MPI job depends on a set of properties contained within a text file. These properties are read into a `Properties` object contained within the `Resources` object.

The core MPI classes (`Distributor` and `Receiver`) use these properties:

Workers Per Node Used by the receiver process to start the required number of workers;

Logsheet URL Use by distributor and receiver processes (and children) to open the log.

The `Logsheet URL` property is optional, and if present all MPI Framework trace messages will be written to the specified logging target. Two types of Uniform Resource Locator schemes are allowed: `file://` and `syslog://`, corresponding to the types of `Logsheet` classes (Section 6.3 on page 16) in the Framework.

Subclasses and other components of the MPI Framework may add properties as needed, usually to the same file as the above properties. Record-based jobs (using `RecordStoreDistributor` and `RecordProcessor`), for example, have these additional properties:

Input Record Store The input record store;

Chunk Size How many record keys or key-value pairs to place into a work package.

For a record store job, an example properties file might be:

```
Input Record Store = test.rs
Chunk Size = 7
Workers Per Node = 3
Logsheet URL = file://mpi.log
```

Applications can add one or more properties to the file as needed. One example would be a URL for a Logsheet used only by the application.

17.7 MPI Runtime

The `Runtime` class is the interface between the application and the MPI runtime environment. The `argv` and `argc` parameters to the `main()` function as passed through to the `Runtime` object, then onto the core Open-MPI functions. The `Runtime` object also sets up a signal handler for the job, and starts the `Distributor` and `Receiver` processes. A method is also provided for the application to abort the MPI job, providing for a somewhat clean shutdown.

On of the key features of an MPI job under the Framework is premature shutdown with minimal loss of work. Three types of exit condition can be set by sending a signal to the distributor, receiver or worker processes.

SIGQUIT Exit when the current work package is exhausted;

SIGINT Exit when the current work item is finished ("quick exit");

SIGTERM Exit immediately ("term exit").

For the normal exit and quick exit cases, a clean shutdown is performed for the distributor, receivers, and all worker processes. For term exit, each worker process is terminated immediately and therefore cannot finish processing the current work item. However, distributors and receivers will shutdown in a clean manner.

Any of the signals can be sent to the distributor process, which then sends messages to the receivers. In addition, if a signal is sent to a receiver, or worker process, only that process (receiver or worker) is affected, but the termination condition is communicated "up" the chain. By selectively sending signals to certain processes, a user can shutdown the entire job (send to the distributor), an entire node (send to the receiver on that node), or a single worker. A worker receiving a signal sends a message back to the receiver. Likewise, a receiver will communicate the shutdown state back to the distributor.

17.8 Logging

In order to aid tracing and debugging of a parallel job, the MPI Framework can be configured to write trace messages to the log storage. These trace messages are logged as debug messages instead of normal entries. The type and location of the log is given to the Framework by using a URL as a property when starting the MPI job (see Section 17.6 on the preceding page).

When the URL for a log is the `file://` type, the MPI Framework will create several log files on the node where it runs. The reason for this is that during `Receiver` processing, one or more worker processes are created in addition to the main receiver process. Each of these processes requires exclusive access to the file-based log sheet in order to avoid conflicts with the log entry commitment. The log files will be named with the property value as a prefix, and the hostname/MPI task number/process ID added as a suffix. For example, if the property is `file://mpijob.log`, a log file might have a name of `mpijob.log-node01-1-12345`.

To aid logging within the application, access to the `Logsheet` opened by the Framework is available via the class whose interface is implemented within the application, `WorkPackageProcessor`, for example.

Two wrapper functions, `MPI::logMessage()` and `MPI::logEntry()`, are provided in order to "safely" log. These functions handle all errors from the `Logsheet` object, and will turn off log message commitment once an error occurs. The Framework and application can continue processing.

17.9 MPI Framework Applications

An application of the MPI Framework is responsible for implementing several functions declared in the Framework, requiring subclassing of the MPI classes. In this section an example application that processes records from a store will be described.

Listing 17.1 shows the header file that declares a subclass of `RecordProcessor`. The `newProcessor()`, `performInitialization()`, and `processRecord()` methods are those required to complete an implementation of `RecordProcessor`. A memory buffer pointer is managed with a smart pointer object.

Listing 17.1: MPI Framework Application Classes

```
class TestRecordProcessor : public BiometricEvaluation::MPI::RecordProcessor {
```

```
public:
        TestRecordProcessor(
            const std::string &propertiesFileName);
        ~TestRecordProcessor();

        std::shared_ptr<BE::MPI::WorkPackageProcessor>
        newProcessor(std::shared_ptr<BE::IO::Logsheet> &logsheet);

        void
        performInitialization(std::shared_ptr<BE::IO::Logsheet> &logsheet);

        void processRecord(const std::string &key);

        void processRecord(
            const std::string &key,
            const BE::Memory::uint8Array &value);

protected:
private:
        std::shared_ptr<BE::IO::Logsheet> _recordLogsheet;
        std::shared_ptr<char> _sharedMemory;
        uint32_t _sharedMemorySize;
};
```

Next, Listing 17.2 shows the implementation of the class methods. In this simple example, each record is acknowledged with a log entry.

Also shown in several of the methods is the use of the `Logsheet` object provided to the application by the Framework, along with wrapper functions, `logMessage()` and `logEntry()`.

The application also creates its own `Logsheet` object in order to separate Framework log messages from the application messages when processing the actual record. In error cases, the Framework log is used in order to keep the set of calls from the Framework to the application in sequence and package processing together.

A common memory buffer is allocated in `performInitialization()` method, and this buffer's pointer is copied to each processing instance in the `newProcessor()` method. Access to this common memory is shown in each `processRecord()` method. The actual memory buffer is not copied because the Framework will invoke the system call `fork()` which results in all memory of the parent process being copied into the child.

Listing 17.2: MPI Framework Application Implementation

```
#include <be_mpi_receiver.h>
#include <be_mpi_recordstoredistributor.h>
#include <be_mpi_runtime.h>

#include "test_be_mpi.h"

using namespace BiometricEvaluation;

static const std::string DefaultPropertiesFileName("test_be_mpi.props");

/*
 * Implementations of the MPI RecordProcessor class interface.
 * Calls the parent constructor to manage the properties file name.
 */
TestRecordProcessor::TestRecordProcessor(
    const std::string &propertiesFileName) :
    RecordProcessor(propertiesFileName)
```

```
{
}

TestRecordProcessor::~TestRecordProcessor()
{
}

/*
 * Factory object: Log our call and set up the shared memory buffer.
 */
void
TestRecordProcessor::performInitialization(
    std::shared_ptr<IO::Logsheet> &logsheet)
{
        this->setLogsheet(logsheet);

        /*
         * Set up the memory that will be shared across all instances.
         */
        char *buf = (char *)malloc(SHAREDMEMORYSIZE);
        strcpy(buf, "I am the god of shared memory!");
        this->_sharedMemorySize = std::strlen(buf);
        this->_sharedMemory = std::unique_ptr<char>(buf);

        *logsheet.get() << std::string(__FUNCTION__) << " called: ";
        *logsheet.get()
            << "Shared memory size is " << this->_sharedMemorySize
            << " and contents is [" << buf << "]";
        BE::MPI::logEntry(*logsheet.get());
}

/*
 * Factory object: Create a new instance of the TestRecordProcess
 * that will work on work package records. Each instance gets
 * its own instance of the log sheet.
 */
std::shared_ptr<BiometricEvaluation::MPI::WorkPackageProcessor>
TestRecordProcessor::newProcessor(
    std::shared_ptr<IO::Logsheet> &logsheet)
{
        std::string propertiesFileName =
            this->getResources()->getPropertiesFileName();
        TestRecordProcessor *processor =
            new TestRecordProcessor(propertiesFileName);
        processor->setLogsheet(logsheet);

        /*
         * If we have our own Logsheet property, and we can open
         * that Logsheet, use it for record logging; otherwise,
         * create a Null Logsheet for these events. We use the
         * framework's Logsheet for tracing of processing, not
         * record handling logs.
         */
        std::string url;
        std::unique_ptr<BE::IO::PropertiesFile> props;
        try {
```

```
                /* It is crucial that the Properties file be
                 * opened read-only, else it will be rewritten
                 * when the unique ptr is destroyed, causing
                 * a race condition with other processes that
                 * are reading the file.
                 */
                props.reset(new BE::IO::PropertiesFile(
                    propertiesFileName, IO::READONLY));
                url = props->getProperty(
                    TestRecordProcessor::RECORDLOGSHEETURLPROPERTY);
        } catch (BE::Error::Exception &e) {
                url = "";
        }
        processor->_recordLogsheet = BE::MPI::openLogsheet(
            url, "Test Record Processing");
        processor->_sharedMemory = this->_sharedMemory;
        processor->_sharedMemorySize = this->_sharedMemorySize;

        std::shared_ptr<BiometricEvaluation::MPI::WorkPackageProcessor> sptr;
        sptr.reset(processor);
        return (sptr);
}

/*
 * Helper function to log some information about a record.
 */
static void
dumpRecord(
    BE::IO::Logsheet &log,
    const std::string key,
    const Memory::uint8Array &val)
{
        log << "Key [" << key << "]: ";
        /* Dump some bytes from the record */
        for (uint64_t i = 0; i < 8; i++) {
                log << std::hex << (int)val[i] << " ";
        }
        log << " |";
        for (uint64_t i = 0; i < 8; i++) {
                log << (char)val[i];
        }
        log << "|";
        BE::MPI::logEntry(log);
}

/*
 * The worker object: Log to the Framework Logsheet, obtain the data for
 * the record, and log some information to the record Logsheet.
 */
void
TestRecordProcessor::processRecord(const std::string &key)
{
        BE::IO::Logsheet *log = this->getLogsheet().get();

        if (this->getResources()->haveRecordStore() == false) {
                BE::MPI::logMessage(*log, "processRecord(" + key + ")"
```

```
                    + " called but have no record store; returning.");
                return;
        }
        *log << "processRecord(" << key << ") called: ";
        char *buf = this->_sharedMemory.get();
        *log << "Shared memory size is " << this->_sharedMemorySize
            << " and contents is [" << buf << "]";
        BE::MPI::logEntry(*log);

        Memory::uint8Array value(0);
        std::shared_ptr<IO::RecordStore> inputRS =
            this->getResources()->getRecordStore();
        try {
                inputRS->read(key, value);
        } catch (Error::Exception &e) {
                *log << string(__FUNCTION__) <<
                    " could not read record: " <<
                    e.whatString();
                return;
        }
        /*
         * Log record info to our own Logsheet instead of
         * the one provided by the framework.
         */
        BE::IO::Logsheet *rlog = this->_recordLogsheet.get();
        dumpRecord(*rlog, key, value);
}

/*
 * The worker object: Log to the Framework Logsheet, and log some record
 * information to the record Logsheet.
 */
void
TestRecordProcessor::processRecord(
    const std::string &key,
    const BiometricEvaluation::Memory::uint8Array &value)
{
        BE::IO::Logsheet *log = this->getLogsheet().get();
        *log << "processRecord(" << key << ", [value]) called: ";
        char *buf = this->_sharedMemory.get();
        *log << "Shared memory size is " << this->_sharedMemorySize
            << " and contents is [" << buf << "]";
        BE::MPI::logEntry(*log);

        /*
         * Log record info to our own Logsheet instead of
         * the one provided by the framework.
         */
        BE::IO::Logsheet *rlog = this->_recordLogsheet.get();
        dumpRecord(*rlog, key, value);
}
```

Listing 17.3: MPI Framework Application Main

```
int
main(int argc, char* argv[])
```

```
{
        /*
         * It is important that the MPI runtime environment be started
         * prior to any other activity that may result in premature
         * termination. Therefore, participate in the MPI environment, but
         * don't create a Receiver or Distributor until any local items
         * are take care of.
         */
        MPI::Runtime runtime(argc, argv);

        std::unique_ptr<MPI::RecordStoreDistributor> distributor;
        std::unique_ptr<MPI::Receiver> receiver;
        std::shared_ptr<TestRecordProcessor> processor;

        /*
         * If there is any argument to the program, use keys only as the
         * record distribution method. Otherwise, use keys and values.
         */
        bool includeValues;
        if (argc == 1) {
                MPI::printStatus("Test Distributor and Receiver, keys only");
                includeValues = false;
        } else {
                MPI::printStatus("Test Distributor and Receiver, keys and values");
                includeValues = true;
        }
        try {
                distributor.reset(
                    new MPI::RecordStoreDistributor(propFile, includeValues));

                processor.reset(new TestRecordProcessor(propFile));

                receiver.reset(new MPI::Receiver(propFile, processor));

                runtime.start(*distributor, *receiver);
                runtime.shutdown();
        } catch (Error::Exception &e) {
                MPI::printStatus("Caught: " + e.whatString());
                runtime.abort(EXIT_FAILURE);
        } catch (...) {
                MPI::printStatus("Caught some other exception");
                runtime.abort(EXIT_FAILURE);
        }

        return (EXIT_SUCCESS);
}
```

References

[1] *ANSI INCITS 378-2004: Finger Minutiae Format for Data Interchange*. ANSI/INCITS, 2004. 39, 43, 50

[2] *ISO/IEC 19794-2: Information technology - Biometric data interchange formats - Part 2: Finger minutiae data*. ISO/IEC, first edition, 2005. 39, 50

[3] *American National Standard for Information Systems - Data Format for the Interchange of Fingerprint, Facial, & Other Biometric Information*. ANSI/NIST-ITL, 1-2007 edition, 2007. 4, 39, 41, 47

[4] Mark Adler. zlib, 2012. `http://www.zlib.net/`. 19

[5] World Wide Web Consortium. Portable Network Graphics Standard, 2003. `http://www.w3.org/TR/PNG/`. 35

[6] Independent JPEG Group. libjpeg, 2011. `http://www.ijg.org/`. 34

[7] Joint Photographic Experts Group. JPEG2000 Image Standard, 1992. `http://www.jpeg.org/jpeg2000/index.html`. 34

[8] Joint Photographic Experts Group. JPEG Image Standard, 2011. `http://www.jpeg.org/jpeg/index.html`. 34

[9] International Committee for Information Technology Standards. `http://www.incits.org`. 43, 50

[10] ISO/IEC Joint Technical Committee 1/SC 37 Biometrics. 43

[11] Communications and Remote Sensing Lab, Université catholique de Louvain. OpenJPEG Library, 2011. `http://www.openjpeg.org/`. 34

[12] Message Passing Interface Forum. `http://www.mpi-forum.org`. 4, 55

[13] NIST Biometric Image Software, 2011. `http://www.nist.gov/itl/iad/ig/nbis.cfm`. 7, 34, 35

[14] NIST Image Group. `http://www.nist.gov/itl/iad/ig`. 1

[15] Greg Roelofs. libpng, 2011. `http://www.libpng.org/pub/png/libpng.html`. 35

[16] Bjarne Stroustrup. *The C++ Programming Language*. Addison Wesley, special edition, 2000. 3, 9

[17] The Syslog Protocol. `http://tools.ietf.org/html/rfc5424`. 18

[18] Wavelet Scalar Quantization Gray-Scale Fingerprint Image Compression Standard, 2010. `https://www.fbibiospecs.org/docs/WSQ_Gray-scale_Specification_Version_3_1_Final.pdf`. 35