Nail to Nail Fingerprint Capture Challenge
Software Test Plan

Last Updated: May 17, 2017
Refer to Page 20 for changes.

Contents

1 Introduction

1.1 Overview

The Nail to Nail (N2N) Fingerprint Capture Challenge aims to improve live and forensic biometric fingerprint recognition by improving N2N fingerprint enrollment capture technology while eliminating the human operator. Participants will provide new N2N fingerprint imaging devices capable of meeting these requirements. The images captured by these devices will be fed to a state-of-the-art commercial fingerprint template generation and fingerprint template identification algorithms in order to assess performance. Alternatively, participants may submit their own algorithms, provided they adhere to this Application Program Interface (API) and test plan.

This document aims to clarify technical details regarding the API, as well as defining methods on how to submit an API implementation.

1.2 Timeline

All API implementation participants must adhere to the timeline below. Additional timelines apply to the submission of information for other portions of the project. All deadlines are available on the project website.

2 Fingerprint Data

2.1 New Imagery

All N2N fingerprint imaging devices are required to save images in JPEG-2000 format. These images will be decompressed by *Biometric Evaluation Framework* and transmitted as defined in Section 3. The participant's software library shall be able to create enrollment and search templates from the decompressed versions of the images created by their N2N fingerprint imaging device.

Complete guidelines on the required image format are available on the project website.

2.2 Background Imagery

The software library shall be required to handle creating enrollment and search templates from a large quantity of "legacy" operational rolled imagery. This imagery will be used to seed an enrollment set on which the software library will search. Images may originate from inked fingerprint card scans or live scan devices.

This background imagery shall be provided to the software library as uncompressed raw 8-bit grayscale, as transmitted according to Section 3. The images will be scanned at 500 or 1 000 pixels per inch horizontal and vertical resolution. Software libraries should call the `getResolution()` method on images and process them into a format suitable for matching, if necessary.

All data provided as background imagery will contain at most one finger. Fingerprint segmentation routines will not be required.

2.3 Latent Imagery

The software library may support searching latent fingerprint images. These images will be cropped to include a single latent fingerprint per image. Some exceptions to this rule may occur due to overlapping prints, but it is expected that automated latent matching software accounts for such inputs. There are no guarantees made regarding the orientation of the fingerprint in the image.

Latent images may differ in the number of channels, bits per channel, and resolution. Software libraries should call the `getColorDepth()`, `getBitDepth()`, and `getResolution()` methods on these images and process them into a format suitable for matching, if necessary.

2.4 Proprietary Data

In addition to a JPEG-2000 image, participants have the option to save a "proprietary" image or template directly from their N2N fingerprint imaging device. The time to create and save this data is included in the recorded physical data acquisition time for the N2N fingerprint imaging device, so it may be advisable to extract

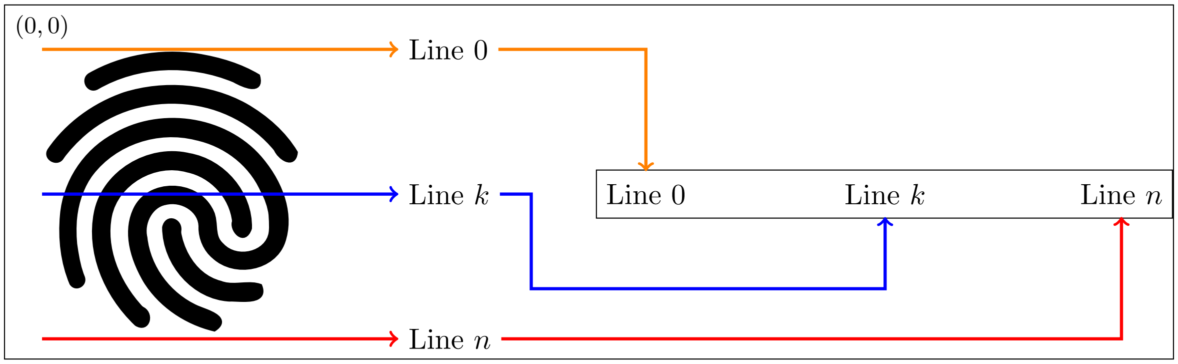

Figure 1: Order of scanned lines.

features using the API. This data will be transmitted "as-is" to the software library during enrollment and search template creation methods.

There are rules and regulations for saving this data to ensure fairness, as defined on the project website. Not adhering these rules may result in rejecting the participant's software library and/or disqualification from the N2N Fingerprint Capture Challenge.

3 Image Data Interchange

The test driver is responsible for decompressing all standard imagery and providing it to the software library. This section defines the format in which images will be transmitted to the software library. The test driver aims to transmit images in a standarized format to prevent participants from needing to implement new image-handling routines.

3.1 Scan Lines

Images shall follow the scan sequence as defined by ISO/IEC 19794-4:2005, §6.2, and paraphrased here. Figure 1 illustrates the recording order for the scanned image. The origin is the upper left corner of the image. The X-coordinate position (horizontal) shall increase positively from the origin to the right side of the image. The Y-coordinate position (vertical) shall increase positively from the origin to the bottom of the image.

3.2 Colors

Ensure that the appropriate values are passed to your software library by ensuring the correct encoding of information in the JPEG-2000 header. A sample program, beImageInfo.cpp, can be used to mimic the decompression of images that *Biometric Evaluation Framework* will perform.

3.2.1 Colorspace/Number of Channels

RGBA (four channels), RGB (three channels), and Grayscale (one channel) colorspaces are permitted for software libraries. When using the RGB colorspace, pixels will be passed with the red channel first, followed by the green channel, and then the blue channel. If an alpha channel is provided, it will immediately follow the blue channel. All channels will have the same numbers of bits when decompressed.

The number of bits per pixel of the image can be determined by dividing the result of an `Image` instance's `getColorDepth()` method by the result of `getBitDepth()`.

3.3 Channel Depth

The number of bits per channel of the image can be determined by calling the `getBitDepth()` method on the `Image` instance. The API does not support channel bit depths other than 8 or 16. If the provided N2N fingerprint imaging device creates images in a lower bit depth, it should pad the channel values to fit within 8 or 16 bits. Bit depths larger than 16 are not supported.

3.3.1 8-bit

The minimum value that will be assigned to a "black" pixel is `0x00`. The maximum value that will be assigned to a "white" pixel is `0xFF`. Intermediate gray levels will have assigned values of `0x01`–`0xFE`. The pixels are stored left to right, top to bottom, with one 8-bit byte per pixel.

The number of bytes in an image is equal to its height multiplied by its width as measured in pixels. The image height and width in pixels will be supplied to the software library as supplemental information.

3.3.2 16-bit

The minimum value that will be assigned to a "black" pixel is `0x00 00`. The maximum value that will be assigned to a "white" pixel is `0xFF FF`. Intermediate gray levels will have assigned values of `0x00 01`–`0xFF FE`. The pixels are stored left to right, top to bottom, with two 8-bit bytes per pixel. Data will be provided in little-endian format, where the most significant bit of the channel's value will be at the 9th of 16 bits. For example, the intermediate value 4 660 (`0x12 34`) would be transmitted as `0b0011 0100 0001 0010`.

The number of bytes in an image is equal to its height multiplied by its width as measured in pixels, multiplied by two. The image height and width in pixels will be supplied to the software library as supplemental information.

4 Testing Procedure

Because of the live data collection portion of the N2N Fingerprint Capture Challenge, the advertised deadlines (Section 1.2) are very strict and cannot be extended. Failure to adhere to these deadlines may result in disqualification.

4.1 Validation

All participants submit their software library via the project's validation mechanism. Any software library submitted in a different manner will be asked to resubmit using the project validation mechanism.

4.2 Timing Test

Upon receiving a validated submission from a participant, a timing test will be run. This test will confirm whether or not the submission meets the timing thresholds defined in Section 6.7. If the submission is not fast enough and there is time prior to the final submission deadline, participants will be asked to speed up their algorithm and resubmit.

Note that not all timing tests may be performed. For example, finalization of a multi-million subject enrolled set may not be feasible until after the submission deadline. Participants should ensure that all timing metrics are met prior to submission.

4.3 "Legacy" Enrollment Template Generation

After timing has been confirmed, a large number of "legacy" rolled fingerprints will be provided to the enrollment template creation method (Section 5.1). This process is expected to be completed for all participants *prior* to the live data collection.

4.4 Record Generation

Once the live data collection has completed, the test facility will collect the images and optional proprietary data from the participant's N2N fingerprint imaging devices, along with the "legacy" rolled captures from "Operator A" and "Operator B." All images will be encoded with subject information in an ANSI/NIST-ITL file.

4.5 N2N Enrollment Template Generation

The Operator B and participant N2N fingerprint imaging device images extracted from the ANSI/NIST-ITL files provided by the test facility will be passed to `makeEnrollmentTemplate()`.

4.6 Enrollment Set Finalization

At this point, all enrollment templates will have been created. Two enrollment sets will be formed by two separate calls to the enrollment set finalization method(Section 5.3). One will be a combination of the "legacy" and "Operator B" imagery, and the other a combination of the "legacy" and participant N2N fingerprint imaging device imagery.

4.7 Search Template Creation

While simultaneously finishing enrollment template creation and enrollment set finalization, search template creation (Section 5.2) will commence in order to create several search templates. It will be necessary to create search templates from "Operator A" and latent imagery.

4.8 Two-Stage Identification

After all enrollment sets have been finalized and all search templates have been created, the two-stage identification process will begin.

4.9 Analysis and Results Notification

Candidate lists and timing will be analyzed after the completion of the two-stage identification. The project sponsor will be notified of the results and will notify the recipients. Only the project sponsor will be able to release results.

4.10 Supplemental Report Generation

Data beyond which participant was the fastest or most accurate will be generated from running the software portion of this N2N Fingerprint Capture Challenge. Supplemental reports detailing this information may be released at the discretion of the project sponsor.

5 API Description

The N2N Fingerprint Capture Challenge software API is broken into three distinct parts:

- Feature Extraction
  - Enrollment Templates
  - Search Templates
- Enrollment Set Generation
- Identification
  - Stage One (partial search)
  - Stage Two (complete search)

The process starts with creating many enrollment and search templates. Next, all enrollment templates are combined by the participant into a proprietary enrollment set, suitable for searching. Finally, search templates are searched against the enrollment set.

It is expected that the total memory needed to store the entire enrollment set in Random Access Memory (RAM) will exceed the RAM capacity of a single test hardware node. For this reason, identification is broken into two stages. During creation of the enrollment set, the `N2N::Interface` implementation is provided the number of nodes that will be available during the first stage of identification. The `N2N::Interface` implementation should use this information to partition the enrollment set into that many parts. Identification calls during the first stage will be provided a number, indicating the enrollment set partition in which to search. The test driver will ensure that each search template will be searched against all partitions. During *Identification: Stage Two*, all information generated from all the partition searches during *Identification: Stage One* will be provided to the `N2N::Interface` implementation simultaneously. This information should be processed and formed into a candidate list returned to the caller.

5.1 Enrollment Template Creation

The software library will be responsible for creating several million enrollment templates. First, the test driver provides an opportunity for the software library to load any configuration files and update the state of the implementation object returned. Next, the test driver `fork`s and repeatedly calls `makeEnrollmentTemplate()`.

This method provides the software library with a variable number of standard images or proprietary N2N fingerprint imaging device data, all from a single subject. The software library should provide a single enrollment template representing the data of the passed-in subject.

A Unified Modeling Language (UML) sequence diagram for this process is provided in Figure 2.

5.1.1 Failure to Extract

Failures to extract shall be reported by the software library returning `ReturnStatus::FailedToExtract` and setting an appropriate value for the enrollment template (likely a 0-byte buffer). All enrollment templates, regardless of the value of `ReturnStatus`, will be provided to the enrollment set generation step described in Section 5.3.

5.2 Search Template Creation

The process of creating search templates is identical to the creation of enrollment templates(Section 5.1). One significant change is that the software library will be told during the initialization step whether or not latent imagery will be provided as sources for feature extraction. It is expected that participants will need to load a different algorithm or set of configuration files to process latent imagery. Only the type of imagery specified during initialization will be provided during the `N2N::Interface` implementation's lifetime.

A UML sequence diagram for this process is provided in Figure 3.

5.2.1 Failure to Extract

Failures to extract shall be reported by the software library returning `ReturnStatus::FailedToExtract` and returning a 0-byte buffer for the search template. These search templates will not be passed to the two-stage identification methods.

5.3 Enrollment Set Creation

Once all enrollment templates have been created, the test driver will provide all enrollment templates to the software library for processing into an enrollment set. The file format of the enrollment set is not specified, but please note that the file system running on the test hardware performs poorly with many small files. Some sort of "database" flat-file format (e.g., BerkeleyDB, SQLite, etc.) is recommended.

The number of test hardware nodes used for *Identification: Stage One* will be provided to the `N2N::Interface` implementation during this process. If necessary for your implementation of identification, ensure that the enrollment set is partitioned into at least this many partitions.

Participants must make a copy of all enrollment templates provided, as it will be their last opportunity to have access to the data in this form.

A UML sequence diagram for this process is provided in Figure 4.

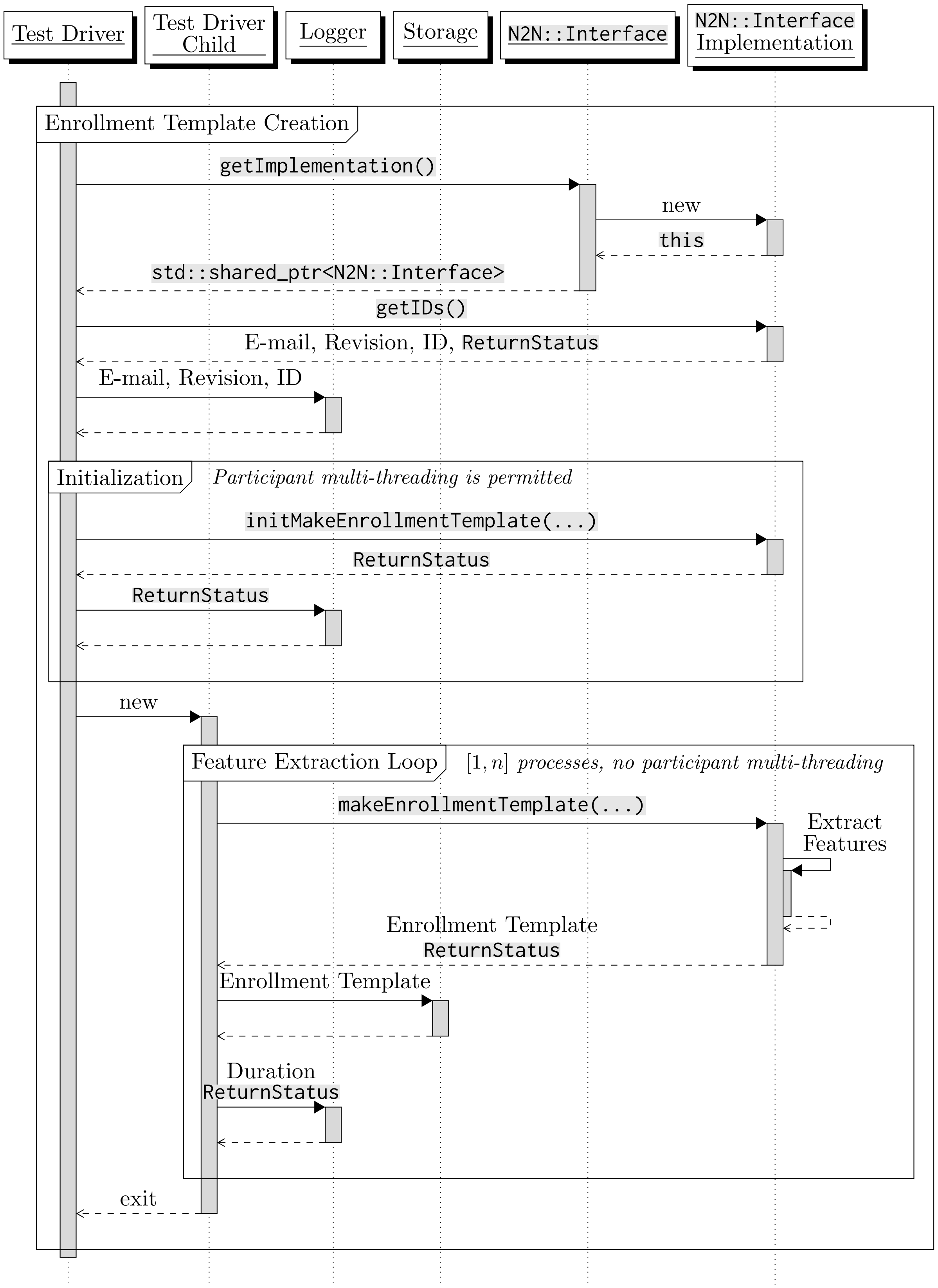
Test Driver
Test Driver Child
Logger
Storage
N2N::Interface
N2N::Interface Implementation
Enrollment Template Creation
getImplementation()
new
this
std::shared_ptr<N2N::Interface>
getIDs()
E-mail, Revision, ID, ReturnStatus
E-mail, Revision, ID
Initialization
Participant multi-threading is permitted
initMakeEnrollmentTemplate(...)
ReturnStatus
ReturnStatus
new
Feature Extraction Loop
[1, n] processes, no participant multi-threading
makeEnrollmentTemplate(...)
Extract Features
Enrollment Template
ReturnStatus
Enrollment Template
Duration
ReturnStatus
exit

Figure 2: Enrollment Template Creation.

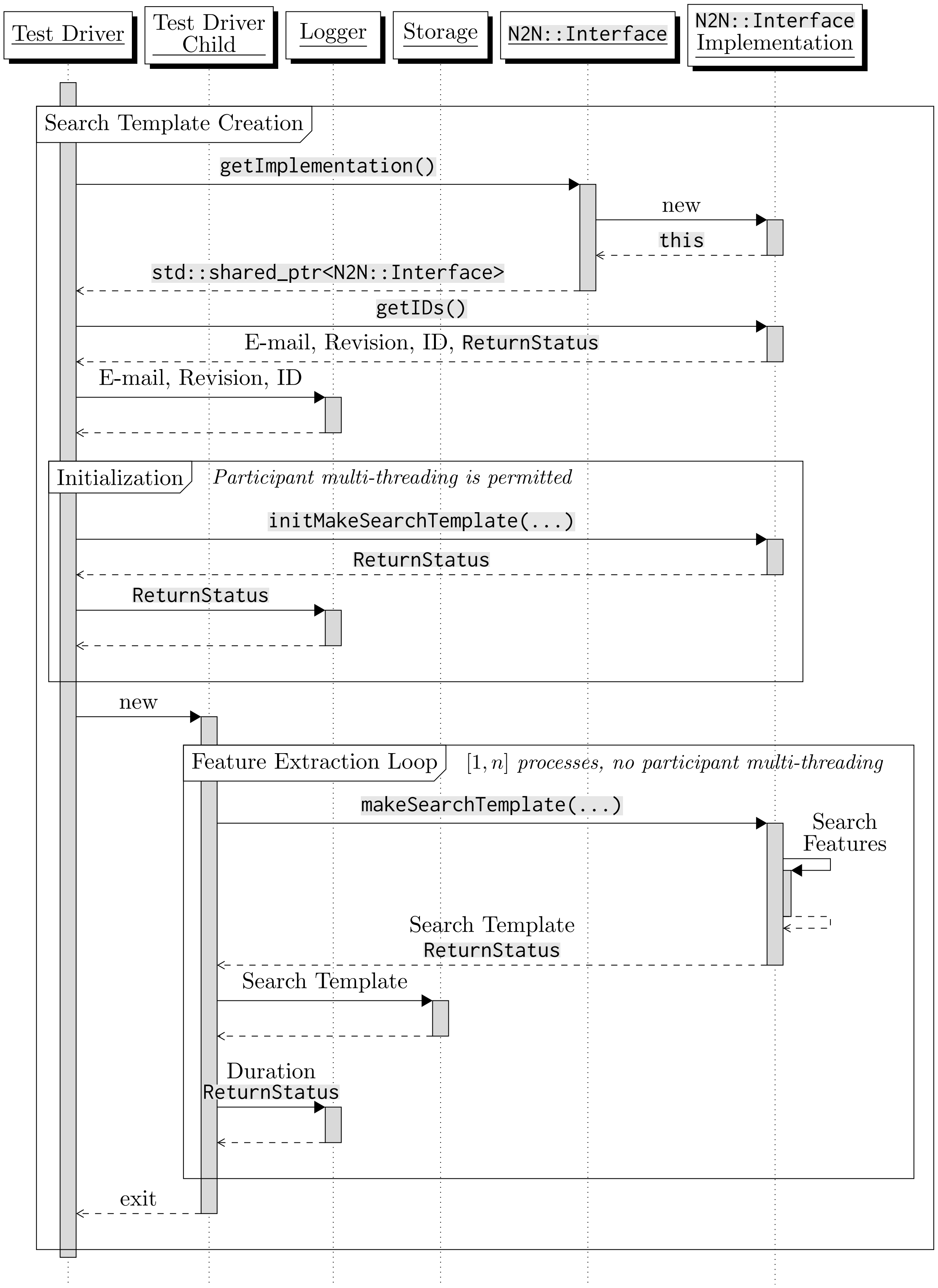

Figure 3: Search Template Creation.

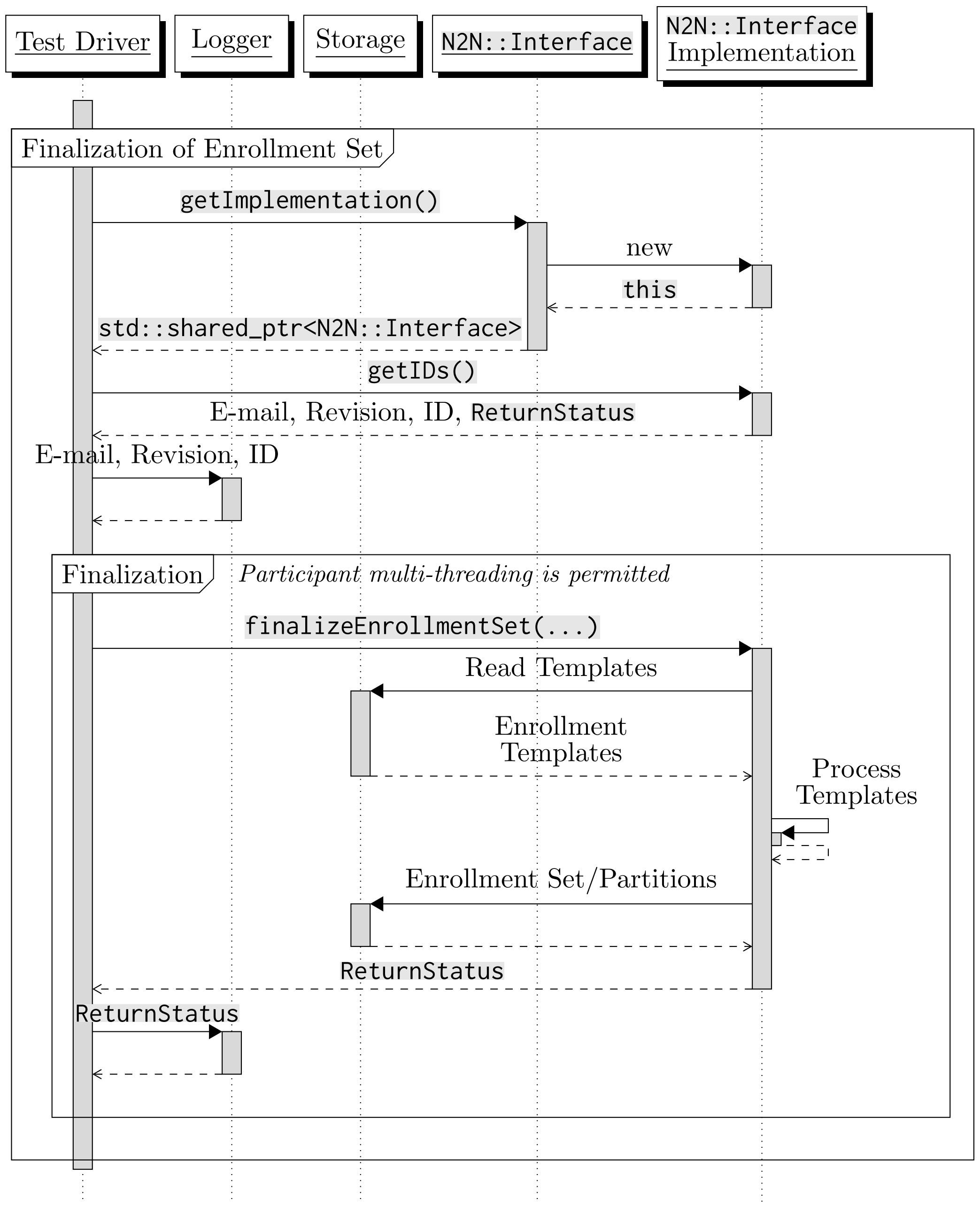
Test Driver
Logger
Storage
N2N::Interface
N2N::Interface Implementation
Finalization of Enrollment Set
getImplementation()
new
this
std::shared_ptr<N2N::Interface>
getIDs()
E-mail, Revision, ID, ReturnStatus
E-mail, Revision, ID
Finalization
Participant multi-threading is permitted
finalizeEnrollmentSet(...)
Read Templates
Enrollment Templates
Process Templates
Enrollment Set/Partitions
ReturnStatus
ReturnStatus

Figure 4: Finalization of Enrollment Set.

The following sequences may occur simultaneously on the same or different physical nodes:

- Node 0:
  - `initIdentificationStageOne(/* ... */, 0);`
  - `identifyTemplateStageOne("search001", tmpl, "/node0RAMDisk/search001")`
    - `N2N::Interface` implementation writes `/node0RAMDisk/search001/results-0.txt`
- Node 1:
  - `initIdentificationStageOne(/* ... */, 1);`
  - `identifyTemplateStageOne("search001", tmpl, "/node1RAMDisk/search001")`
    - `N2N::Interface` implementation writes `/node1RAMDisk/search001/results-1.txt`
- Node 2:
  - `initIdentificationStageOne(/* ... */, 2);`
  - `identifyTemplateStageOne("search001", tmpl, "/node2RAMDisk/search001")`
    - `N2N::Interface` implementation writes `/node2RAMDisk/search001/results-2.txt`

After all calls to `identifyTemplateStageOne()` for a given search ID have completed, the test driver will merge the contents of the individual stage one data directories into a single directory that will be passed to `identifyTemplateStageTwo()`. In this example, the contents of that directory is:

- `results-0.txt`
- `results-1.txt`
- `results-2.txt`

Figure 5: Example of stage one data directory contents.

5.4 Identification: Stage One

A search template, enrollment set partition number, and temporary storage area known as the *stage one data directory* are provided to the `N2N::Interface` implementation. The `N2N::Interface` implementation should search the search template for candidates within the enrollment set partition indicated, and record any necessary intermediate information in the stage one data directory. The format and layout of such intermediate information is not mandated, however all searches running on a single node will have at most 4 GB of storage space combined to store such information. The test driver will periodically move files from this 4 GB shared storage area to a more permanent location, freeing up space. The stage one data directory will be a temporary location in a RAM disk to reduce the effects of I/O operations on the timing requirements for this stage.

The stage one data directory will initially be empty. Once all stage one searches on all enrollment set partitions (over multiple nodes) have completed, the test driver will merge the contents of stage one data directories for a given search ID into a *single* directory. This means that for a given search ID, all files written in the stage one data directory must have a unique filename, or risk being overwritten during the merge. This can be accomplished by appending the enrollment set partition number to any file written during this identification stage. Refer to Figure 5 for an example.

A UML sequence diagram for this process is provided in Figure 6.

5.4.1 Failure to Extract

If a search template failed to extract, it will not be passed to this method.

5.4.2 Failure to Search

If a failure occurs while searching during this stage, an appropriate error code should be returned. The `N2N::Interface` implementation is *still* responsible for recording information about the search in the specified area for use in *Identification: Stage Two*, if desired. All Search IDs passed to *Identification: Stage One* will be passed to *Identification: Stage Two*, regardless of failure. It is the `N2N::Interface` implementation's descision whether or not to return an empty candidate list from *Identification: Stage Two* as a result of a failure in *Identification: Stage One*.

5.5 Identification: Stage Two

After a search template has been searched against all enrollment set partitions in *Identification: Stage One*, the same search template will be provided to stage two of identification. During this stage, the test driver will provide all information written during *Identification: Stage One* for this search template. The `N2N::Interface` implementation should coalesce this information and/or perform more searching over all partitions and return a candidate list. *Identification: Stage Two* will occur in a separate process with a separate instantiation of of the `N2N::Interface` implementation.

A UML sequence diagram for this process is provided in Figure 7.

5.5.1 Candidate List

A maximum of 100 candidates may be returned. There is no minimum number of candidates. The number of candidates in the candidate list will be determined by calling the `size()` method of `vector`, and so all entries in the candidate list must be valid candidates (not "placeholder" objects). Candidates in the list must be sorted by descending similarity score, where `candidate.at(0)` has the highest probability of being a match.

5.5.2 Failure to Extract

If a search template failed to extract, it will not be passed to this method. The test driver will automatically record an empty candidate list.

6 Software and Documentation

The functions specified in this API shall be implemented exactly as defined in the software library. The header file used in the test driver is provided on the project website and as part of the validation package.

6.1 Restrictions

Participants shall provide binary code in the form of a software library only (i.e., no source code or headers). Software libraries must be submitted in the form of a dynamic/shared library file (i.e., ".so" file). This library shall be known as the "core" library, and shall be named according to the guidelines in Section 6.3. Static libraries (i.e., ".a" files) are not allowed. Multiple shared libraries are permitted if technically required and are compatible with the validation package. Any required libraries that are not standard to CentOS 7.2.1511 or installed on the

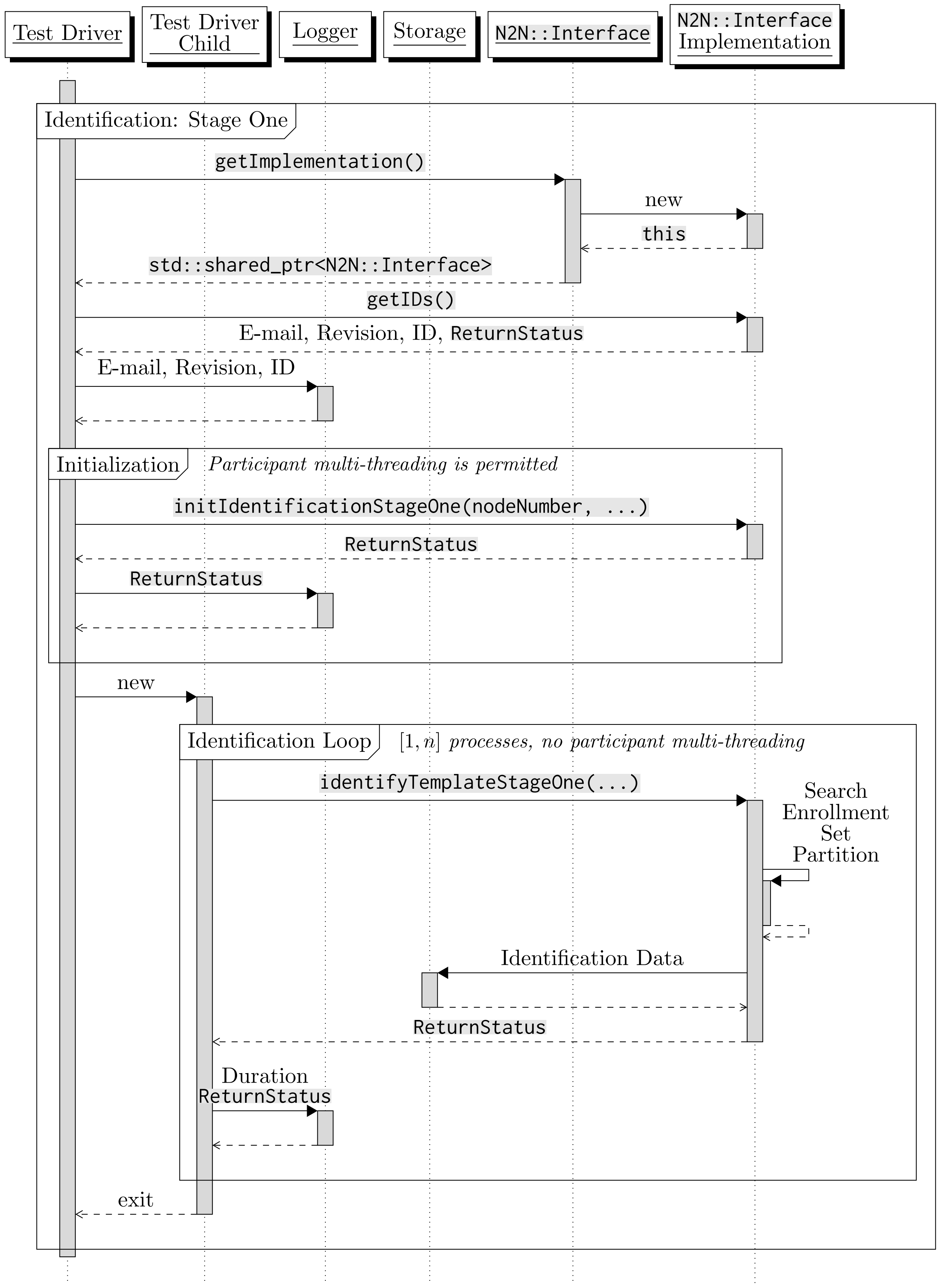
Test Driver
Test Driver Child
Logger
Storage
N2N::Interface
N2N::Interface Implementation
Identification: Stage One
getImplementation()
new
this
std::shared_ptr<N2N::Interface>
getIDs()
E-mail, Revision, ID, ReturnStatus
E-mail, Revision, ID
Initialization
Participant multi-threading is permitted
initIdentificationStageOne(nodeNumber, ...)
ReturnStatus
ReturnStatus
new
Identification Loop
[1, n] processes, no participant multi-threading
identifyTemplateStageOne(...)
Search Enrollment Set Partition
Identification Data
ReturnStatus
Duration ReturnStatus
exit

Figure 6: Identification: Stage One.

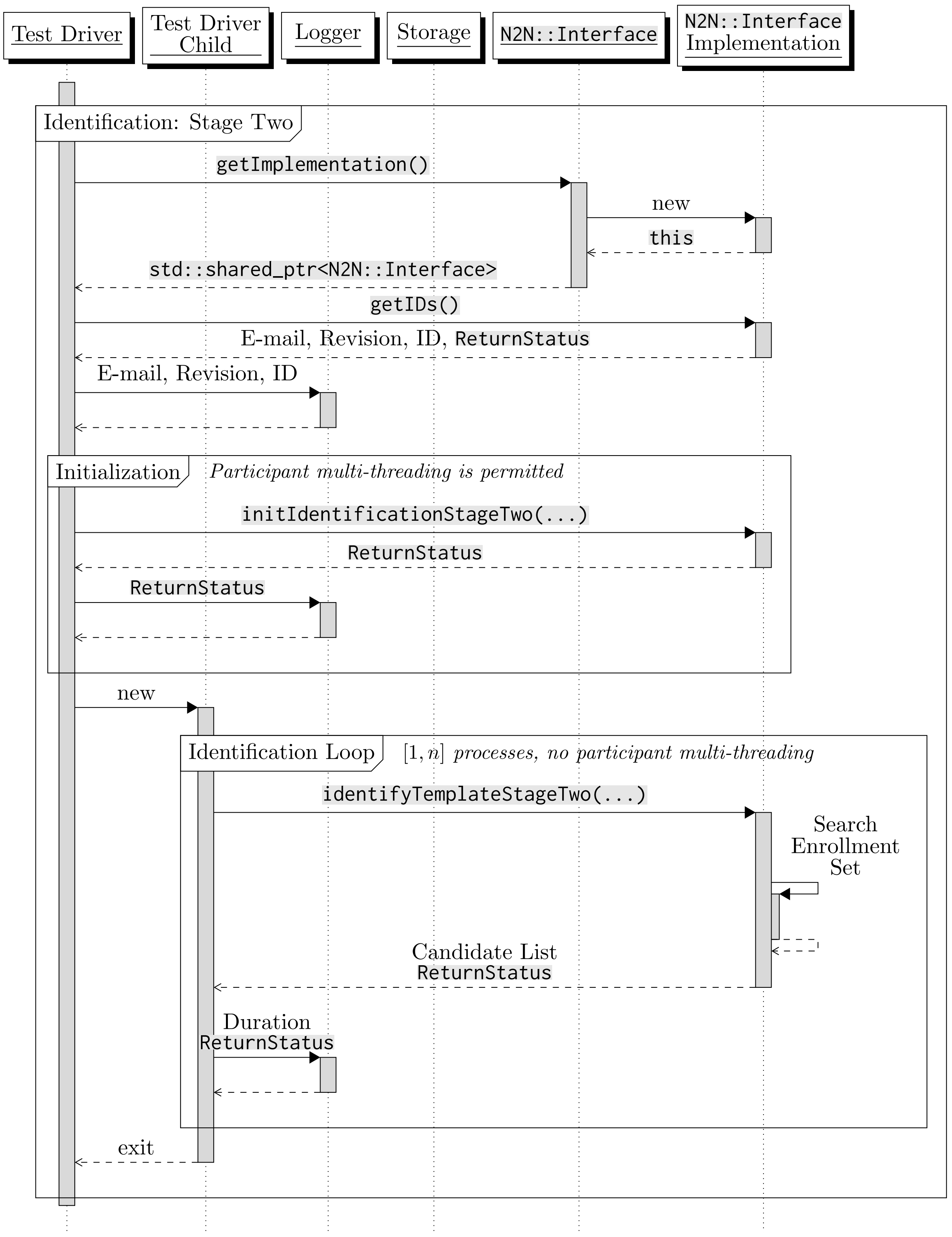
Test Driver
Test Driver Child
Logger
Storage
N2N::Interface
N2N::Interface Implementation
Identification: Stage Two
getImplementation()
new
this
std::shared_ptr<N2N::Interface>
getIDs()
E-mail, Revision, ID, ReturnStatus
E-mail, Revision, ID
Initialization
Participant multi-threading is permitted
initIdentificationStageTwo(...)
ReturnStatus
ReturnStatus
new
Identification Loop
[1, n] processes, no participant multi-threading
identifyTemplateStageTwo(...)
Search Enrollment Set
Candidate List
ReturnStatus
Duration
ReturnStatus
exit

Figure 7: Identification: Stage Two.

testing hardware must be built and submitted alongside the "core" library. All submitted software libraries will be placed in a single directory, and this directory will be added to the runtime library search path list.

Individual software libraries provided must not include multiple modes of operation or algorithm variations. No switches or options will be tolerated within one library. Access to a configuration directory of configuration files is provided, and if necessary, such algorithm configuration changes would occur during the call to an appropriate `init` method.

The software library shall not make use of threading, forking, or any other multiprocessing techniques, *except during calls to* `finalizeEnrollment()` *and* `init` *methods*. The test driver operates as a Message Passing Interface (MPI) job over multiple compute nodes, and then forks itself into many processes. In the test environment, there is no advantage to threading. It limits the usefulness of NIST's batch processing and makes it impossible to compare timing statistics across participants. During `finalizeEnrollment()` and `init` methods, up to $C - 1$ threads may be used for threading. This value should be queried at runtime using `std::thread::hardware_concurrency`.

The software library shall remain stateless. It shall not acknowledge the existence of other processes running on the test hardware, such as through `semaphore`s or `pipe`s. It shall not acknowledge the existance of other machines on the network. It shall not write to arbitrary file system locations. The only file system writes should occur during calls to API methods requiring file system writes. It shall not attempt network requests.

6.2 External Dependencies

It is preferred that the API specified by this document be implemented in a single "core" library. Additional libraries may be submitted that support this "core" library file (i.e., the "core" library file may have dependencies implemented in other libraries if a single library is not feasible).

Use of compiler and architecture optimizations is allowed and strongly encouraged, but software libraries must be able to run on the following processor types:

- AMD Opteron 8376HE
- Intel Xeon E5405, X5680, X5690, X7560
- Intel Xeon E5-2680

6.3 Naming

The "core" software library submitted shall be named in a predefined format. The first part of the software library's name shall be `libN2N_`. The second piece of the software library's name shall be a case-sensitive unique identifier that has been provided to you by the test administrator, followed by an underscore. The final part of the software library's name shall be a version number, followed by the file extension `.so`. Supplemental libraries may have any name, but the "core" library must be dependent on supplemental libraries in order to be linked correctly. The **only** library that will be explicitly linked to the test driver is the "core" library, as demonstrated in Section 6.4.

The identifier and version number provided in the file name must match the values output by `getIDs()`. The version number must be incremented with every subsequent submission. With this naming scheme, **every "core" library received shall have a unique filename**. Incorrectly named or versioned software libraries will be rejected.

Example
Company X requests to participant in this software evaluation by submitting their public signing and encryption key. The project sponsor replies with the identifier "01F3" for use in Company X's `N2N::Interface` implementation. Company X submits a software library named `libN2N_01F3_1.so` with revision 1 of their algorithm. This library assigns `01F3` to the `identifier` out parameter and `1` to the `revision` out parameter of `getIDs()`. It is determined that Company X's validation fails and the software library is rejected. Company X submits revision 2 to correct the defects in 1. Company X updates the value of `revision` in their implementation of `getIDs()` to `2` and renames their library to `libN2N_01F3_2.so`.

6.4 Operating Environment

The software library will be tested in non-interactive "batch" mode (i.e., without terminal support) in an isolated environment (i.e., no Internet or other network connectivity). Thus, the software library shall not use any interactive functions, such as graphical user interface calls, or any other calls that require terminal interaction (e.g., writes to `stdout`).

NIST will link the provided library files to a C++ language test driver application using the compiler `g++` (version `4.8.5-4`, via `mpicxx`) under **CentOS 7.2.1511**. For example:

```
mpicxx -std=c++11 -o N2N N2N.cpp -Llib -lN2N_01FE_2
```

Participants are required to provide their software libraries in a format that is linkable using `g++` with the test driver. All compilation and testing will be performed on 64-bit hardware running CentOS 7.2.1511. Participants are strongly encouraged to verify library-level compatibility with `g++` on CentOS 7.2.1511 **prior to** submitting their software to avoid unexpected problems.

6.5 Usage

The software library shall be executable on any number of machines without requiring additional machine-specific license control procedures, activation, hardware dongles, or any other form of rights management.

The software library usage shall be **unlimited**. No usage controls or limits based on licenses, execution date/time, number of executions, etc., shall be enforced by the software library.

6.6 Validation and Submitting

A *validation package* that will link the participant's "core" software library to a sample test driver will be provided. Once the validation successfully completes on the participant's system, a file with validation data and the participant's software library will be created. After being signed and encrypted, **only** this signed and encrypted file shall be submitted to N2NChallenge@nist.gov. Any software library submissions not generated by an unmodified copy of the N2N Fingerprint Capture Challenge validation package will be rejected. Any software library submissions that generate errors while running the validation package on the test hardware will be rejected. Validation packages that have recorded errors while running on the participant's system will be rejected. Any software library submissions not generated with the current version of the validation package will be rejected. Any submissions of successful validation runs not created on CentOS 7.2.1511 will be rejected.

This validation step is crucial to determine that the software library submitted by participants is operating correctly and as expected in different hardware environments. Any difference in candidate lists or similarity scores on the test hardware as compared to the participant's hardware, regardless of the number of nodes used

Method	Maximum Time	Measurement
`getIDs()`	0 seconds	Exact
`initMakeEnrollmentTemplate()`	5 minutes	Exact
`makeEnrollmentTemplate()`	3 seconds	90th percentile
`finalizeEnrollment()`	90 minutes per 1 million subjects	Exact
`initMakeSearchTemplate()`	5 minutes	Exact
`makeSearchTemplate()`	3 seconds	90th percentile
`initIdentificationStageOne()`	5 minutes	Exact
`initIdentificationStageTwo()`	5 minutes	Exact
`identifyTemplateStageOne()` + `identifyTemplateStageTwo()`	1 minute	90th percentile

Table 1: Speed requirements for API methods.

during the first stage of identification, will not be tolerated. Differences in search and enrollment templates on the test hardware and participant hardware are permitted, but strongly discouraged.

6.7 Speed

Timing tests will be run and reported on a sample of the “legacy” dataset prior to completing the entire test. Submissions that do not meet the timing requirements on the dataset sample will be rejected. Participants may resubmit a faster submission immediately.

7 Biometric Evaluation Framework

Classes from NIST’s *Biometric Evaluation Framework* will be used when writing software to the N2N Fingerprint Capture Challenge API. These classes are designed to be easy to learn and use. Methods and concepts that will be necessary during implementation are detailed in this section.

Participants are free to use other parts of *Biometric Evaluation Framework* in their implementations. There are two restrictions necessary to ensure that the test driver, which relies on *Biometric Evaluation Framework*, continues to work correctly. Participants shall not modify the source code of the library itself (though if you find a bug, please feel free to submit an issue). Participants shall also not import identical symbol definitions into their library. The test driver will be linked against `libbiomeval.so.9.0`, which will include all symbols necessary for participant software libraries.

7.1 BiometricEvaluation::Finger

`/usr/local/include/be_finger.h`

The `Finger` namespace contains enumerated constants that will be used to transmit information about the fingers represented in transmitted imagery.

Notable enumerations include:

- `Finger::Position`: Finger positions, such as `RightThumb` and `LeftIndex`. The underlying numeric values correspond to the finger position codes (FGP) from ANSI/NIST-ITL.
- `Finger::Impression`: Impression type of the captured fingerprint.
  - **NOTE**: For this test, all N2N images will be represented by `LiveScanRolled`. There will be no differentiation between "legacy" rolled imagery and the participant's N2N fingerprint imaging device imagery.
  - **NOTE**: `N2N::Interface` implementations will be notified during initialization whether to prepare for latent imagery. It is expected that a different algorithm or configuration within the core software library will be needed when processing latent imagery.

7.2 BiometricEvaluation::Image

`/usr/local/include/be_image.h`
`/usr/local/include/be_image_image.h`

The `Image::Image` class encapsulates information about standardized images through classes defined in the `Image` namespace. Images provided to the `N2N::Interface` implementation will be decompressed and encapsulated into an `Image::Raw`.

Notable methods include:

- `getColorDepth()`: Obtain the number of bits per pixel.
- `getBitDepth()`: Obtain the number of bits per channel.
- `getResolution()`: Obtain the scanning resolution.
  - The values are stored in `xRes` and `yRes` of the resulting object.
- `getDimensions()`: Obtain the image dimensions, in pixels.
  - The values are stored in `xSize` and `ySize` of the resulting object.
- `getRawData()`: Obtain a buffer of decompressed raw bytes.
  - **NOTE**: For `Image::Raw`, `getData()` will return the same information as `getRawData()`.

7.3 BiometricEvaluation::IO::RecordStore

`/usr/local/include/be_io_recordstore.h`

When creating the finalized enrollment set (Section 5.3), enrollment templates will be provided to the `N2N::Interface` implementation through an `IO::RecordStore`. These objects are key/data-pair storage abstractions that are designed for high-performance on the test hardware filesystems.

Individual key/data-pair elements of an `RecordStore` are stored in an `IO::Record` struct, with `key` and `data` members. Dereferencing an `IO::RecordStores`'s iterator will provide an `IO::Record`.

Notable methods include:

- `getCount()`: Obtain the number of key/data pairs.

- `sequence()`: Obtain the “next” key/data pair, until exhausted.
- `begin()/end()`: Iterate through all key/data pairs.

7.4 BiometricEvaluation::Error::Exception

`/usr/local/include/be_error_exception.h`

Should any truly exceptional conditions happen in your `N2N::Interface` implementation, throw an instance of an `Error::Exception`. An optional `std::string` can be included with additional information.

`N2N::Interface` implementations should not throw an exception as a substitute for returning a failure condition. Exceptions should only be thrown in situations where a non-recoverable error has occurred.

7.5 BiometricEvaluation::Memory::uint8Array

`/usr/local/include/be_memory_autoarray.h`

A `Memory::uint8Array` is a specialization of `Memory::AutoArray` for `uint8_t`. It is used in many places in the API and `Biometric Evaluation Framework` for managing dynamic memory buffers. It provides the convenience and C compatibility of a `uint8_t*` with the benefits of `std::vector<uint8_t>`.

`uint8Array`s are *implicitly* convertible to `uint8_t*` and can be used anywhere a `uint8_t*` is required, such as `std::memcpy`.

Notable methods include:

- `resize()`: Grow or shrink the array.
- `size()`: Obtain the number of bytes in the array.
- `operator[]`: Read or write directly from an offset without bounds checking.
- `begin()/end()`: Iterate through all elements.

Disclaimer

Revisions

The latest version of this document can always be found on the GitHub page, `https://github.com/usnistgov/IARPA-N2N`.

- **17 May 2017**
  - Updates to background imagery resolutions and new information about latent imagery (Section 2).
- **17 February 2017**
  - Added example of stage one data directory merging (Section 5.4 and Figure 5).
- **26 January 2017**
  - First release.