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

NIST Special Publication 500-288
Revision 0

DRAFT 0

Specification for WS-Biometric Devices Revision 0 Draft 0

Ross J. Micheals
Matt Aronoff
Kayee Kwong
Kevin Mangold
Karen Marshall

NIST Special Publication 500-288
Revision 0

Specification for WS-Biometric Devices (Draft 0)

Recommendations of the National Institute of Standards and Technology

Ross J. Micheals
Matt Aronoff
Kayee Kwong
Kevin Mangold
Karen Marshall

INFORMATION TECHNOLOGY

Biometric Clients Lab
Image Group
Information Access Division
Information Technology Laboratory
National Institute of Standards and Technology
Gaithersburg, MD 20899-8940

Spring 2011

US Department of Commerce
Gary Locke, Secretary

National Institute of Standards and Technology
Patrick D. Gallagher, Director

1 Table of Contents

2 Introduction

Imagine an intelligent biometric device that is cryptographically secure, tamper-proof, and spoof resistant. Such a device would enable biometrics as a viable option for remote authentication. Imagine a new generation of fingerprint scanner, small enough and thin enough to clip onto a police officer's uniform—enabling law enforcement to quickly identify suspects.

These envisioned devices require a communications protocol that is secure, globally connected, and free from requirements on operating systems, device drivers, form factors, and low-level communications protocols. WS-Biometric Devices is a protocol designed in the interest of furthering this goal, with a specific focus on the single process shared by all biometric systems—*acquisition*.

2.1 Terminology

This section contains terminology commonly used throughout this document. First time readers are encouraged to skip this section and revisit it, as needed, after reading §3, Design Concepts and Architecture.

biometric capture device

a system component capable of capturing biometric data in digital form

client

a logical endpoint that originates operation requests

HTTP

Hypertext Transfer Protocol. Unless specified, the term HTTP may refer to either HTTP as defined in [] or HTTPS as defined in [].

sensor or **biometric sensor**

a single biometric capture device, or, a logical collection of biometric capture devices.

target sensor or **target biometric sensor**

the biometric sensor exposed by a particular service

payload parameter or **payload**

an operation parameter that is passed to a service within the content of an HTTP request

URL parameter

2.2 Documentation Conventions

The following documentation conventions are used throughout this document.

2.2.1 Quotations

If the inclusion of a period within a quotation might lead to ambiguity as to whether or not the period should be included in the quoted material, the period will be placed outside the trailing quotation mark. For example, a sentence that ends in a quotation would have the trailing period "inside the quotation, like this quotation punctuated like this.” However, a sentence that ends in a URL would have the trailing period outside the quotation mark, such as “`http://example.com`”.

2.2.2 Machine-Readable Code

With the exception of some reference URLs, Machine readable information will typically be depicted with a `mono-spaced font, such as this.`

2.2.3 Sequence Diagrams

Throughout this document, sequence diagrams are used to help explain various scenarios. These diagrams are informative simplifications and are intended to help explain core specification concepts. Operations are depicted in a functional, remote procedure call style. The level of abstraction presented in the diagrams, and the details that are shown (or not shown) will vary according to the particular information being illustrated.

2.3 Normative References

[RFC2119]	S. Bradner, *Key words for use in RFCs to Indicate Requirement Levels*, *http://www.ietf.org/rfc/rfc2119.txt*, IETF RFC 2119, March 1997.
[RFC2616]	R. Fielding, et al. Hypertext Tranfer Protocol—HTTP/1.1, *http://www.ietf.org/rfc/rfc2616.txt*, IETF RFC 2616, June 1999.
[RFC4122]	P. Leach, M. Mealling, and R. Salz, *A Universally Unique Identifier (UUID) URN Namespace, http://www.ietf.org/rfc/rfc4122.txt*, IETF RFC 4122, July 2005.
[XSDPart1]	*XML Schema Part 1: Structures Second Edition*, *http://www.w3.org/TR/xmlschema-1*, W3C Recommendation. 28 October 2004.
[XSDPart2]	*XML Schema Part 2: Datatypes Second Edition*, *http://www.w3.org/TR/xmlschema-2*, W3C Recommendation. 28 October 2004.

2.4 Informative References

3 Design Concepts and Architecture

This section describes the major design concepts and overall architecture of WS-Biometric Devices (WS-BD). The main purpose of a WS-Biometric Devices service is to expose a target biometric sensor to clients via web services.

3.1 Interoperability

ISO/IEC 2382-1 (1993) defines *interoperability* as "*the capability to communicate, execute programs, or transfer data among various functional units in a manner that requires the user to have little to no knowledge of the unique characteristics of those units"*

Conformance to a standard does not necessarily guarantee interoperability. *There should be an example here of where this happens by design.*

A major design goal of WS-Biometric Devices (WS-BD) is to *maximize* interoperability, by *minimizing* the amount of required "knowledge of the unique characteristics" of a component that supports WS-Biometric Devices (WS-BD).

3.2 Architectural Components

Before discussing the envisioned use of WS-Biometric Devices, it may be useful to distinguish between the various components that might comprise a WS-Biometric Devices implementation. These are *logical* components, and may or may not correspond to particular *physical* boundaries. This distinction becomes vital in understanding WS-Biometric Devices's operational models.

3.2.1 Clients

A *client* is any software component that originates requests for biometric acquisition. Note that a client might be one of many hosted in a parent (logical or physical) component, and that a client might originate requests to a variety of destinations.

This icon is used to depict an arbitrary WS-Biometric Devices client. A personal digital assistant (PDA) is used to serve as a reminder that a client might be hosted on a non-traditional computer.

3.2.2 Sensor

A biometric *sensor* is any component that is capable of acquiring, i.e., digitally sampling, a biometric. Most sensor components are hosted within a dedicated hardware component, but this is not necessarily globally true. For example, a keyboard is a general input device, but might also be used for a keystroke dynamics biometric.

This icon is used to depict a biometric device. The icon has a vague similarity to a fingerprint scanner, but should be thought of as an arbitrary biometric sensor.

As discussed in §1.1, the term “sensor” in used in this document in a singular sense, but may in fact be referring to multiple biometric capture devices.

3.2.3 Sensor Service

The *sensor service* is the “middleware” software component that exposes a biometric sensor to a client through web services. The sensor service adapts HTTP request-response operations to biometric sensor command & control.

This icon is used to depict a sensor service. The icon is abstract and has no meaningful form, just as a sensor service is a piece of software that has no physical form.

3.3 Intended Use

Each implementation of WS-Biometric Devices will be realized via a mapping of logical to physical components. A distinguishing characteristic of an implementation will be the physical location of the sensor service component. WS-Biometric Devices is designed to support two scenarios.

- The sensor service and biometric sensor are hosted by different physical components. A *physically separated service* is one where there is both a physical and logical separation between the biometric sensor and the service that provides access to it.

- The sensor service and biometric sensor are hosted within the same physical component. A *physically integrated service* is one where the biometric sensor, and the service that provides access to it, reside within the same physical component.

Figure 1 depicts a physically separated service. In this scenario, a biometric sensor tethered to a personal computer, workstation, or server. The web service, hosted on the computer, listens for communication requests from clients. An example of such an implementation would be a USB fingerprint scanner attached to a personal computer. A lightweight web service, running on that computer could listen to requests from local (or remote) clients—translating WS-Biometric Devices requests to and from biometric sensor commands.

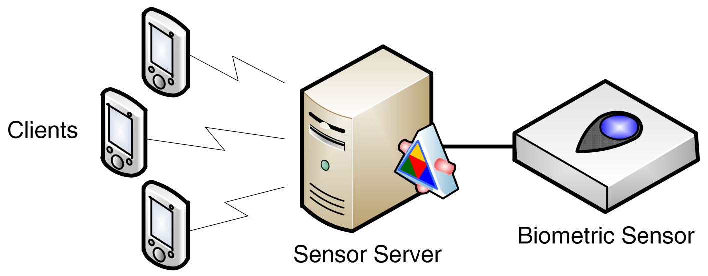

Figure 1. A physically separated WS-Biometric Devices (WS-BD) implementation.

Figure 2 depicts a physically integrated service. In this scenario, a single hardware device has, embedded within it, a biometric sensor, as well as a web service. Similar functionality is seen in many network printers; it is possible to point a web browser to a local network address, and obtain a web page that displays information about the state of the printer, such as toner and paper levels. Clients make requests directly to the integrated device; and an web service running within an embedded system translates the WS-Biometric Devices requests to and from biometric sensor commands.

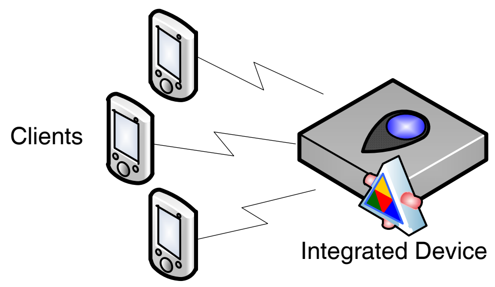

Figure 2. A physically integrated WS-Biometric Devices implementation.

Finally, it should be admitted that from a systems engineering viewpoint, the “separate” versus “integrated” distinction is indeed a simplification with a high degree of fuzziness. For example, one might imagine putting a hardware shell around a USB fingerprint scanner connected to a small form-factor computer. Inside the shell, the sensor service and sensor are on different physical components. Outside the shell, the sensor service and sensor appear integrated. The definition of what constitutes the “same” physical component depends on the particular implementation, and the intended level of abstraction. Regardless, it is a useful distinction in that it illustrates the flexibility possible afforded by leveraging highly interoperable communications protocols.

3.4 General Service Behavior

The following section describes the general behavior of WS-Biometric Devices clients and services.

3.4.1 Security Model

In this version of the protocol it is assumed that if a client is able to establish a HTTP (or HTTPS) communication with the sensor service, then the client is fully authorized to use the service. This implies that all successfully connected clients have equivalent authority may be considered *peers*.

Clients might be prevented by connected through various HTTP protocols, such as HTTPS with client-side certificates, or a more sophisticated protocol such as OpenId (http://openid.net/) and/or OAuth.

Recommended security profiles are discussed in Appendix 0.

3.4.2 HTTP Request-Response Usage

Most biometrics devices are inherently *single user*—i.e., they are designed to sample the biometrics from a single user at a given time. Web services, on the other hand, are inherently designed to be *stateless* and *multiuser.* A biometric device exposed via web services must therefore provide a mechanism to reconcile these competing designs.

Notwithstanding the native limits of the underlying web server , WS-Biometric Devices services *must* be capable of handling multiple, concurrent requests.) Services *must* respond to requests for operations that do not require exclusive control of the biometric sensor without blocking until the biometric sensor is in a particular state.

Because there is no well-accepted mechanism for providing asynchronous notification via REST, each individual operation *must* block until completion. Clients are not expected to poll—rather make a single HTTP request and block for the corresponding result. Because of this, it is expected that a client would perform WS-Biometric Devices operations on an independent thread, so not to interfere with the general responsiveness of the client application. WS-Biometric Devices clients therefore *must* be configured in such a manner such that individual HTTP operations have timeouts that are compatible with a particular implementation.

WS-Biometric Devices operations may be longer than typical REST services. Consequently, there is a clear need to differentiate between service level errors and HTTP communication errors. WS-Biometric Devices services *must* pass-through the status codes underlying a particular request. In other words, services *must not* use (or otherwise 'piggyback') HTTP return codes to indicate service-level failure. If a service receives a well-formed request, then the service MUST return an 'OK' response code (200). Service failures are described within the contents of the XML data returned to the client for any given operation. *There may need to be a clarification about how the set configuration can return 400 when it receives malformed (as in malformed XML) requests.*

This is deliberately different from REST services that override HTTP return codes to provide service-specific error messages. This design patterns facilitates clearer communication of failures and errors than overriding particular HTTP failure codes, and, provides better support for HTTP libraries that do not easily support HTTP return code overrides.

3.4.3 Client Identity

Before discussing how WS-Biometric Devices balances single-user vs. multi-user needs, it is first necessary to understand the WS-Biometric Devices model for how an individual client can easily and consistently identify itself to a service.

HTTP is, by design, a *stateless* protocol. Therefore, any persistence about the originator of a sequence of requests must be built in (somewhat) artificially to the layer of abstraction above HTTP itself. This is accomplished in WS-Biometric Devices via *session*—a collection of operations that originate from the same logical endpoint. To initiate a session, a client performs a *registration* operation and obtains a *session identifier* (or "session id"). During subsequent operations, a client uses this id as a parameter to uniquely identify itself to a server. When the client is finished, it is expected to close a session with an *unregistration* operation. To conserve resources, services may unregister clients that do not explicitly unregister after a period of inactivity (see §5.4.2.1).

This use of a session id directly implies that the particular sequences that constitute a session are entirely the responsibility of the *client*. A client might opt to create a single session for its entire lifetime, or, might open (and close) a session for a limited sequence of operations. WS-Biometric Devices supports both scenarios.

It is not recommended, but it is possible, that a client might maintain multiple sessions with the same service simultaneously. For simplicity, this documentation will be written under the assumption that a single client maintains either up to one session. This can be assumed without loss of generality, since a client with multiple sessions to a service could be decomposed into to "sub-clients"—one (sub-)client per session id.

3.4.4 Locking

WS-Biometric Devices uses a *lock* to satisfy two complimentary requirements:

1. A service must have exclusive, sovereign control over biometric sensor hardware to perform a particular *sensor operation* such as initialization, configuration, or capture.
2. A client needs to perform an uninterrupted sequence of sensor operations.

Each WS-Biometric Devices service exposes a *single* lock (one per service) that controls access to the sensor. Clients obtain the lock in order to perform a sequence of operations that should not be interrupted. Obtaining the lock is an indication to the server (and, indirectly to peer clients) that a (1) a series of sensor operations is about to be initiated and (2) that server may assume sovereign control of the biometric sensor.

A client releases the lock upon completion of its desired sequence of task. This indicated to the server (and, again, indirectly to peer clients) that the uninterruptable sequence of operations is finished. A client might obtain and release the lock many times within the same session, or, a client might open a close a session for each pair of lock/unlock operations. This decision is entirely dependent on particular client.

The statement that a client might "own" or "hold" a lock is a convenient simplification that makes it easier to reason or speak about a client-server interaction. In reality, each sensor service maintains a unique global variable that contains a session id. The originator of that session id can be thought of as the client that "holds" the lock. Therefore, the other phrase used is that a service is locked *to* a particular id.

Clients are expected to release the lock after completing their required sensor operations, but there is a mechanism for locks from ill-behaved clients to be broken. This feature is necessary to ensure that a client cannot hold a lock indefinitely, and deny its peers access to the biometric sensor.

As stated previously, it is implied that all successfully connected clients enjoy the same access privileges. This is critically important, because it is this implied equivalence of "trust" that affords a lock *stealing* operation.

3.4.5 Operations Summary

All WS-Biometric Devices operations fall into one of eight categories.

1. Registration
2. Locking
3. Metadata
4. Initialization
5. Configuration
6. Capture
7. Download
8. Cancellation

Of these, the *initialization, configuration, capture,* and *cancellation* operations are all *sensor operations* (i.e., they require exclusive sensor control) and require locking. *Registration, locking,* and *download* are all **non**-sensor operations. They do not require locking and (as stated earlier) must be available to clients regardless of the status of the biometric sensor. There are two *metadata* operations, one that is a sensor operation, one that is not. This allows a client to obtain information that must be obtained from the sensor hardware itself, such a firmware version.

The following is a brief summary of each type of operation:

- *Registration* operations open and close (unregister) a session.
- *Locking* operations are used by a client to obtain the lock, release the lock, and *steal* the lock away from a misbehaved client.
- *Metadata* operations query the service for information about the service itself, such as the supported biometric modalities, and common service configuration parameters.
- The *initialization* operation prepares the biometric sensor for operation.
- *Configuration* operations get or set sensor parameters.
- The *capture* operation performs the acquisition of the biometrics.
- *Download* operations transfer the captured biometric data from the service to the client.
- Sensor operations can be stopped by the *cancellation* operation.

Note that *download* is *not* a sensor operation. This allows for a collection of clients to dynamically share biometric data. One client might perform the capture and hand off download responsibility to a peer.

3.4.6 Idempotency

The OASIS Reference Model for Service Oriented Architecture (1.0) define idempotentcy as

> *A characteristic of a service whereby multiple attempts to change a state will always and only generate a single change of state if the operation has already been successfully completed once.*

In this document, we extend this definition by adding the caveat that no other operations may occur in between. Idempotent operations *may* have side-effects—but the final state of the service must be the same over multiple (uninterrupted) invocations.

> **EXAMPLE**: A service has an available lock. A client invokes the lock operation and obtains a “success” result. A subsequent invocation of the operation also returns a “success” result. Even though the service changed from an unlocked to a locked state, because the end results of the two sequential operations were identical (the service was locked and the client received a “success” result), the call is idempotent.

To best support robust communications, WS-Biometric Devices has been designed to offer idempotent services whenever possible.

3.4.7 Service Lifecycle Behavior

It is significantly easier for a physically separated implementation to emulate the behavior of a fully integrated implementation than it is the other way around. For instance, on a desktop computer, hot-swapping the target biometric sensor is possible through an operating system’s plug-and-play architecture. However, changing the target biometric sensor out of a handheld device is probably beyond the capabilities of most modern devices. Therefore, the lifecycle of a service *must* behave as if it was an *integrated* implementation, even if it is a separated implementation. The lifecycle of the web server component *must* match the lifecycle of the target biometric sensor.

Fortunately, since HTTP is a connectionless protocol, a client may not be directly affected by a service restart, if the service is written in a robust manner. For example, upon detecting a new target biometric sensor, a robust server could *quiese* (refusing all new request until any pending requests are completed) and automatically restart.

Upon restarting, services *should* return to a fully reset state—i.e., all sessions should be dropped, and the lock *should not* have an owner. A high-availability service *may* have a mechanism to preserve state across restarts, but this is not recommended. A client that communicated with a service that was restarted would lose both its session and the service lock (if it held it). With the exception of the *get common info* operation, through various fault statuses a client would receive indirect notification of a service restart. If needed, a client could use service's common metadata timestamp (§*forward reference needed*) to detect potential changes in the *get common info* operation.

4 Data Dictionary

This section contains descriptions of the data elements that comprise WS-Biometric Devices's type system. Each data type is described via an accompanying XML Schema fragment [XSDPart1, XSDPart2]. For brevity, the XML namespace prefix `xs` represents the namespace "`http://www.w3.org/2001/XMLSchema`". Definitions for the `xs` data types (i.e., those not explicitly defined here) can be found in [XSDPart2].

The target namespace for the data types defined here is "`http://nist.gov/itl/bws/ws-bd/L1/r0/`".

It should be assumed that each data type is defined in the target namespace "`http://nist.gov/itl/bws/ws-bd/L1/r0/`". The XML namespace prefix `wsbd` is shorthand for this namespace.

Data types UUID, WsbdDictionary, WsbdStringArray, and WsbdUuidArray are generic types without any particular semantic meaning.

4.1.1 UUID

A UUID is a unique identifier as defined in [RFC4122]. A service *must* use UUIDs that conform to the following XML Schema fragment.

```
<xs:simpleType name="uuid">
 <xs:restriction base="xs:string">
  <xs:pattern value="[\da-fA-F]{8}-[\da-fA-F]{4}-[\da-fA-F]{4}-[\da-fA-F]{4}-[\da-fA-F]{12}"/>
  </xs:restriction>
</xs:simpleType>
```

EXAMPLE: Each line in the following code fragment contains a well-formed UUID. *These examples should be vetted for correctness.*

```
E47991C3-CA4F-406A-8167-53121C0237BA
10fa0553-9b59-4D9e-bbcd-8D209e8d6818
161FdBf5-047F-456a-8373-D5A410aE4595
```

4.1.2 WsbdDictionary

A WsbdDictionary is a generic container used to hold an arbitrary collection of name-value pairs.

```
<xs:complexType name="WsbdDictionary">
  <xs:sequence>
    <xs:element minOccurs="0" maxOccurs="unbounded" name="item">
      <xs:complexType>
        <xs:sequence>
          <xs:element name="key" nillable="true" type="xs:string" />
          <xs:element name="value" nillable="true" type="xs:anyType" />
        </xs:sequence>
      </xs:complexType>
    </xs:element>
  </xs:sequence>
</xs:complexType>
```

WsbdDictionary instances are nestable. *There should be a note about the use of xs:anyType, and how conformance is (or is not?) effected by the use of types outside of those not documented here.*

4.1.3 WsbdStringArray

A WsbdStringArray is a generic container used to hold a collection of strings.

```
<xs:complexType name="WsbdStringArray">
    <xs:sequence>
      <xs:element minOccurs="0" maxOccurs="unbounded" name="element" nillable="true"
type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
```

EXAMPLE: Each line in the following code fragment is an example of a valid WsbdStringArray. Enclosing tags (which may vary) are omitted. *These examples should be vetted/validated against the schema for correctness.*

```
<element>sessionId</element>
<element>value1</element><element>value2</element>
<element>leftThumb</element><element>rightThumb</element>
```

4.1.4 WsbdUuidArray

A WsbdUuidArray is a generic container used to hold a collection of UUIDs..

```
<xs:complexType name="WsbdUuidArray">
    <xs:sequence>
      <xs:element minOccurs="0" maxOccurs="unbounded" name="element" nillable="true"
type="wsbd:uuid" />
    </xs:sequence>
  </xs:complexType>
```

EXAMPLE: The following code fragment is an example of a *single* WsbdUuidArray with three elements. Enclosing tags (which may vary) are omitted. *This example should be vetted/validated against the schema for correctness.*

```
<element>E47991C3-CA4F-406A-8167-53121C0237BA</element>
<element>10fa0553-9b59-4D9e-bbcd-8D209e8d6818</element>
<element>161FdBf5-047F-456a-8373-D5A410aE4595</element>
```

4.1.5 WsbdStatus

The WsbdStatus represents a common enumeration for communicating state information about a service.

```
<xs:simpleType name="WsbdStatus">
    <xs:restriction base="xs:string">
      <xs:enumeration value="Success" />
      <xs:enumeration value="Failure" />
      <xs:enumeration value="InvalidId" />
      <xs:enumeration value="Canceled" />
      <xs:enumeration value="CanceledWithSensorFailure" />
      <xs:enumeration value="SensorFailure" />
      <xs:enumeration value="LockNotHeld" />
      <xs:enumeration value="LockHeldByAnother" />
      <xs:enumeration value="SensorNeedsInitialization" />
      <xs:enumeration value="SensorNeedsConfiguration" />
      <xs:enumeration value="SensorBusy" />
      <xs:enumeration value="SensorTimeout" />
      <xs:enumeration value="Unsupported" />
      <xs:enumeration value="BadValue" />
      <xs:enumeration value="NoSuchParameter" />
      <xs:enumeration value="PreparingDownload" />
    </xs:restriction>
    </xs:simpleType>
```

4.1.5.1 Definitions

The following table defines all of the potential values for the WsbdStatus enumeration.

Value	Description
success	The operation completed successfully.
failure	The operation failed. The failure was due to a web service (as opposed to a *sensor* error).
invalidId	The provided id is not valid. This can occur if the client provides a (session or capture) id that is either: (a) unknown to the server (i.e., does not correspond to a known registration or capture result), or (b) the session has been marked inactive because too much time has passed between operations associated with the provided (session) id
canceled	The operation was cancelled. NOTE: A sensor service might cancel its own operation if it is taking too long. This can happen if a service maintains its own internal timeout that is shorter than a sensor timeout.
canceledWithSensorFailure	The operation was cancelled, but during (and perhaps because of) cancellation, a sensor failure occurred. This particular status accommodates for hardware that may not natively support cancelation.
sensorFailure	The operation could not be performed because a biometric sensor (as opposed to web service) failure. **NOTE**: Clients that receive *sensorFailure should* assume that the sensor will need to be reinitialized in order to restore normal operation.
lockNotHeld	The operation could not be performed because the client does not hold the lock. **NOTE**: This status implies that at the time the lock was queried, no other client currently held the lock. Therefore, this is not a guarantee that any subsequent attempts to obtain the lock will succeed.
lockHeldByAnother	The operation could not be performed because another client currently holds the lock.
initializationNeeded	The operation could not be performed because the sensor requires initialization.
configurationNeeded	The operation could not be performed because the sensor requires configuration.
sensorBusy	The operation could not be performed because the sensor is currently performing another task.
sensorTimeout	The operation was not performed because the biometric sensor experienced a timeout. **NOTE**: The most common cause of a sensor timeout would be a lack of interaction with a sensor within an expected timeframe.
unsupported	The service does not support the requested operation.

	NOTE: Typically, this status occurs because the sensor was configured into a state that was syntactically valid, but not supported.
badValue	The operation could not be performed because a value provided for a particular parameter was either (a) an incompatible type or (b) outside of an acceptable range.
noSuchParameter	The operation could not be performed because the service did not recognize the name of a provided parameter.
preparingDownload	The operation could not be performed because the service is currently preparing captured data for download.

There should be a general note here identifying those status values that are more relevant for physically separate implementations, and how an physically integrated service/sensor would be less likely to use some of the various return values.

4.1.5.2 Parameter Failures

Services MUST distinguish among `invalidId`, `noSuchParameter`, and `badValue` according to the following rules:

1. If an operation does not recognize a provided parameter *name*, then the service *must* return *noSuchParameter*.
2. If an operation recognizes a provided parameter name, but cannot accept the provided *value*, the then service *must* return *badValue*.
3. If an operation recognizes a URL parameter as an RFC-compliant UUID (i.e., the UUID is parseable), but cannot accept the provided value, then the service *must* return an *invalidId.*

(Informative) It may be helpful to think of *invalidId* as a special case of *badValue* reserved for URL parameters of type UUID.

4.1.5.3 Precedence of Status Enumerations

To maximize the amount of information given to a client when an error is obtained, all WS-Biometric Devicesservices *must* return status values according to the same well-defined priority. In other words, when multiple status messages might apply, a higher-priority status *must* always be returned in favor of a lower-priority status.

The status priority, listed from highest priority ("success") to lowest priority ("failure") is as follows:

1. `success`
2. `invalidId`
3. `noSuchParameter`
4. `badValue`
5. `unsupported`
6. `canceledWithSensorFailure`
7. `canceled`
8. `lockHeldByAnother`
9. `lockNotHeld`
10. `sensorBusy`
11. `sensorFailure`
12. `sensorTimeout`
13. `initializationNeeded`
14. `configurationNeeded`
15. `preparingDownload`
16. `failure`

What follows are two examples illustrating some of the consequences of this ordering. These may be easier to understand after §5.

EXAMPLE: Figure 3 illustrates that client cannot receive a "sensorBusy" status if it does not hold the lock, even if a sensor operation is in progress.

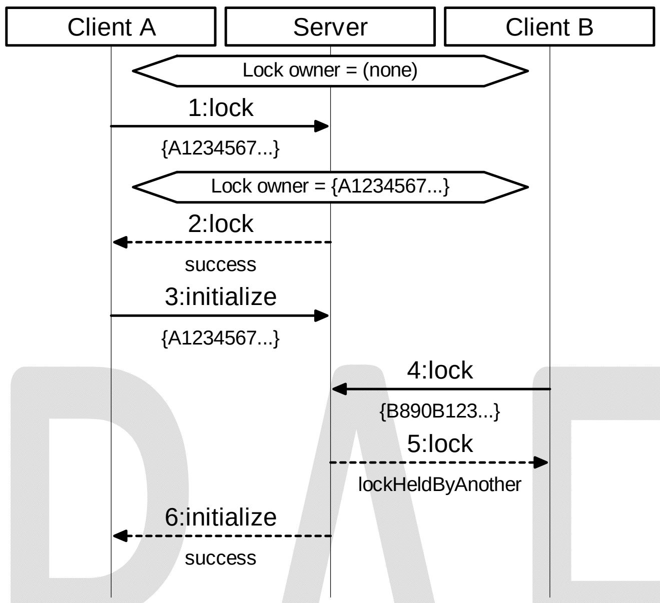

Figure 3. Example illustrating how a client cannot recieve a "sensorBusy" status if it does not hold the lock.

Suppose there are two clients; Client A and Client B. Client A holds the lock and starts initialization (Step 1–3). Immediately after Client A initiates capture, Client B (Step 4) tries to obtain the lock while Client A is still capturing. In this situation, the valid statuses that could be returned to Client B are "sensorBusy" (since the sensor is busy performing a capture) and "lockHeldByAnother" (since Client A holds the lock). In this case, the service returns "lockHeldByAnother" (Step 5) since "lockHeldByAnother" is higher priority than "sensorBusy."

EXAMPLE: The status "canceledWithSensorFailure" and "canceled" are deliberately higher priority than "lockHeldByAnother" and "lockNotHeld." This is to cover the situation where a client steals a lock, and cancels an operation already in progress.

4.2 WsbdResult

All WS-Biometric Devices operations *must* return a WsbdResult that conforms to the following XML Schema fragment.

```
<xs:complexType name="WsbdResult">
 <xs:sequence>
  <xs:element minOccurs="1" name="status" type="wsbd:WsbdStatus"/>
  <xs:element minOccurs="0" name="badFields " nillable="true" type="wsbd:WsbdStringArray"/>
  <xs:element minOccurs="0" name="sessionId" nillable="true" type="wsbd:uuid"/>
  <xs:element minOccurs="0" name="captureIds " nillable="true" type="wsbd:WsbdUuidArray"/>
  <xs:element minOccurs="0" name="commonInfo" nillable="true" type="wsbd:WsbdDictionary"/>
  <xs:element minOccurs="0" name="configuration" nillable="true" type="wsbd:WsbdDictionary"/>
  <xs:element minOccurs="0" name="detailedInfo" nillable="true" type="wsbd:WsbdDictionary"/>
  <xs:element minOccurs="0" name="message" nillable="true" type="xs:string"/>
```

```
    <xs:element minOccurs="0" name="mimeType" nillable="true" type="xs:string"/>
    <xs:element minOccurs="0" name="sensorData" nillable="true" type="xs:base64Binary"/>
  </xs:sequence>
  </xs:complexType>
```

The following is a brief informative description of each WsbdResult element.

Element	Description
`status`	The disposition of the operation. All WsbdResults *must* contain a `status` element.
`badFields`	The list of fields that contain invalid or ill-formed values
`sessionId`	A unique session identifier (§5.3, §5.4)
`commonInfo`	Service metadata that is independent of session and does not require control of the target biometric sensor (§5.8
`detailedInfo`	Service metadata that is session dependent, or requires control of the target biometric sensor (§5.9
`configuration`	The target biometric sensor's current configuration (§5.11, §5.12)
`captureIds`	Identifiers that may be used to obtain data acquired from a capture operation (§5.13, §5.15)
`contentType`	The format of the biometric for the corresponding capture identifier (§5.14)
`sensorData`	The biometric data corresponding to a particular capture identifier (§5.15)

The next major section describes the use of each WsbdResult element in detail.

5 Operations

This section provides detailed information regarding each WS-Biometric Devices operation.

5.1 General Usage Notes

The following usage notes apply to all operations, unless the detailed documentation for a particular operation conflicts with these general notes, in which case the detailed documentation takes precedence.

1. **Failure messages are informative.** If an operation fails, then the message element *may* contain an informative message regarding the nature of that failure. The message is for informational purposes only—the functionality of a client *must not* depend on the contents of the message.
2. **Results must only contain required and optional elements.** Services *must only* return elements that are not required or optional. All other elements must *not* be contained in the result, even if they are empty elements. Likewise, to maintain robustness in the face of a non-conformant service, clients *should* ignore any element that is not in the list of permitted WsbdResult elements for a particular operation call.
3. **Sensor operations must not occur within a non-sensor operation.** Services *must not* perform any sensor control within operations [LIST]. Similarly, services are permitted (but not required) to perform operations that control the biometric sensor *only* within the following operations:
4. **Sensor operations must require locking.** Even if a service implements a sensor operation without controlling the target biometric sensor.
5. **Content Type.** Clients *must* make HTTP requests using the `application/xml` content type.

5.1.1 Visual Summaries

The following two tables provide a visual summary of WS-Biometric Devices operations.

5.1.1.1 Input & Output

The following table represents a visual summary of the various inputs and outputs corresponding to each operation.

URL	HTTP Method	Operation	Payload	Idempotent	Sensor Operation	Status	Bad Field(s)	SessionId	Info	Configuration	Capture Id(s)	MIME Type	Sensor Data	Section
register	POST	register	none			●		●						5.3
register/{sessionId}	DELETE	unregister	none	◆		●	●							5.4
lock/{sessionId}	POST	try lock	none	◆		●	●							5.5
lock/{sessionId}	PUT	steal lock	none	◆		●	●							5.6
lock/{sessionId}	DELETE	unlock	none	◆		●	●							5.7
info	GET	get common info	none	◆		●			●					5.8
info/{sessionId}	GET	get detailed info	none	◆	■	●	●		●					5.9
initialize/{sessionId}	POST	initialize	none	◆	■	●	●							5.10
configure/{sessionId}	GET	get configuration	none	◆	■	●	●			●				5.11
	POST	set configuration	config	◆	■	●	●							5.12
capture/{sessionId}	POST	capture	none		■	●	●				●			5.13
capture/{captureId}	GET	get content type	none	◆		●	●					●		5.14
download/{captureid}	GET	download	none	◆		●	●					●	●	5.15
download/{captureid}/{maxSize}	GET	thrifty download	none	◆		●	●						●	5.16
cancel/{sessionId}	POST	cancel operation	none	◆	■	●	●							5.17

Presence (absence) of a symbol in a table cell indicates that the operations are idempotent (not idempotent), a sensor operation (not a sensor operation) and which elements of the WsbdResult are required.

EXAMPLE: The *capture* operation is not idempotent, a sensor operation, and may contain the elements `status`, `badFields`, and/or `captureIds` in its WsbdResult. The specific rules regarding which elements are permitted depending on the

5.1.1.2 Permitted Status Values

The following table provides a visual summary of the status values permitted.

Operation Description	Success	Failure	Invalid Id	Cancelled	Cancelled with Sensor Failure	Sensor Failure	Lock Not Held	Lock Held By Another	Sensor Needs Initialization	Sensor Needs Configuration	Sensor Busy	Sensor Timeout	Unsupported	Bad Value	No Such Parameter	Preparing Download
register	●	●														
unregister	●	●	●								●			●		
try lock	●	●	●					●						●		
steal lock	●	●	●											●		
unlock	●	●	●					●						●		
common metadata	●	●														
detailed metadata	●	●	●	●	●	●	●	●	●		●	●		●		
initialize	●	●	●	●	●	●	●	●			●	●		●		
get configuration	●	●	●	●	●	●	●	●	●	●	●	●		●		
set configuration	●	●	●	●	●	●	●	●	●		●	●	●	●	●	
capture	●	●	●	●	●	●	●	●	●	●	●	●		●		
get content type	●	●	●											●		●
download captured data	●	●	●											●		●
thrifty download	●	●	●										●	●		●
cancel	●	●	●				●	●						●		

The presence (absence) symbol in a cell indicates that the respective status may (may not) be returned by the corresponding operation.

EXAMPLE: The *register* operation may only return a WsbdResult with a WsbdStatus that contains either `success` or `failure`. The *unregister* operation may only return `success`, `failure`, `invalidId`, `sensorBusy`, or `badValue`.

5.2 Documentation Conventions

Each WS-Biometric Devices operation is documented according to the following conventions.

5.2.1 General Information

Each operation begins with the following tabular summary:

Description	A short description of the operation
HTTP Method	The HTTP method that triggers the operation, i.e., `GET`, `POST`, `PUT`, or `DELETE`
URL Template	The suffix used to access the operation. These take the form

	`resourceName`
	 or
	 `resourceName/{URL_parameter1}/…/{URL_parameter_N}`
	 Each parameter, `{URL_parameter_N}` *must* be replaced, in-line with the value to be passed in for that particular parameter.
	
	 Parameters have no explicit names, other than defined by this document or reported back to the client within the contents of a `badFields` element.
	
	 It is assumed that consumers of the service will prepend the URL to the service endpoint as appropriate.
	
	 EXAMPLE: The resource `resourceName` hosted at the endpoint
	 `http://example.com/Service`
	 would be accessible via
	 `http://example.com/Service/resourceName`

URL Parameters	A description of the URL-embedded operation parameters. For each parameter the following details are provided: • the name of the parameter • the expected data type (§16) • a description of the parameter
Payload	A description of the content, if any, to be posted to the service as input to an operation.
Idempotent	Yes—the operation is idempotent (§3.4.6), or, No—the operation is not idempotent
Sensor Operation (Lock Required)	Yes—the service may require exclusive control over the target biometric sensor. No—this operation does not require a lock. No Given the concurrency model (§3.4.4) this value doubles as documentation as to whether or not a lock is required

5.2.2 WsbdResult Summary

5.2.3 Usage Notes

5.2.4 Unique Knowledge

For each operation, there is a brief description of whether or not the operation affords an opportunity for the server or client to exchange information that might otherwise hinder interoperability.

5.2.5 Return Values Detail

5.3 Register

Description	Open a new client-server session
HTTP Method	`POST`
URL Template	`/register`
URL Parameters	None
Payload	Not applicable
Idempotent	No
Sensor Operation	No

5.3.1 WsbdResult Summary

`success`	`status="success"` `sessionId=`session id (UUID)
`failure`	`status="failure"` `message*=`informative message describing failure

5.3.2 Usage Notes

Register provides a unique identifier that can be used to associate a particular client with a server. In a sequence of operations with a service, a *register* operation is likely one of the first operations performed by a client ('Common Info' being the other). It is expected (but not required) that a client would perform a single registration during that client's lifetime.

DESIGN NOTE: By using an UUID, as opposed to the source IP address, a server can distinguish among clients sharing the same originating IP address (i.e., multiple clients on a single machine, or multiple machines behind a firewall).

5.3.3 Unique Knowledge

The *register* operation affords no opportunity to provide or obtain unique knowledge.

5.3.4 Return Values Detail

The *register* operation *must* return a WsbdResult according to the following constraints.

5.3.4.1 Success

Status Value	`success`
Condition	The service accepts the registration request

Required Elements	`status`
	must be populated with the WsbdStatus literal `"success"`
	`sessionId`
	must be populated with a UUID that can be used for subsequent operations.

Optional Elements	None

The "register" operation *must not* provide a `sessionId` of 00000000-0000-0000-0000-000000000000 (i.e., an empty UUID).

5.3.4.2 Failure

Status Value	`failure`
Condition	The service cannot accept the registration request
Required Elements	`status`

	must be populated with the literal `"failure"`
Optional Elements	`message` An informative description of the nature of the failure.

Registration might fail if there are too many sessions already registered with a service. The `message` element *must* only be used for informational purposes. Clients *must not* depend on particular contents of the message element to control client behavior.

There should be a cross-reference to the common metadata section on how a client can determine the maximum number of concurrent sessions a service can support. The common info might also report the number of registered clients, so that as this count approaches.

5.4 Unregister

Description	Close a client-server session
HTTP Method	`DELETE`
URL Template	`/register/{sessionId}`
URL Parameters	`{sessionId}` (UUID, §4.1.1) Identity of the session to remove
Payload	Not applicable
Idempotent	Yes
Sensor Operation	No

5.4.1 WsbdResult Summary

`success`	`status="success"`
`failure`	`status="failure"` `message*`=informative message describing failure
`sensorBusy`	`status="sensorBusy"`
`badValue`	`status="badValue"`, `badFields={"sessionId"}`

5.4.2 Usage Notes

Unregister closes a client-server session. Although not strictly necessary, clients *should* unregister from a service when it is no longer needed. Services *should* support (on the order of) thousands of concurrent sessions, but this cannot be guaranteed, particularly if the service running within limited computational resources. Conversely, clients *should* assume that the number of concurrent sessions that a service can support is limited. *There should be a cross-reference here regarding how a client can determine the number of total vs. current concurrent sessions offered by a service.*

5.4.2.1 Inactivity

A service *may* automatically unregister a client after a period of inactivity, or if demand on the service requires that least-recently used sessions be dropped. This is manifested by a client receiving a status of `invalidId` without a corresponding unregistration. Services *should* set this value on or about the order of 100 minutes. *There should be a cross-reference here regarding how a client can determine the period of inactivity that would trigger automatic closing of a session.*

5.4.2.2 Sharing Session Ids

A session id is not a secret, but clients that share session ids run the risk of having their session prematurely terminated by a rouge peer client. This behavior is permitted, but discouraged. See §3.4.1 for more information about security.

5.4.3 Unique Knowledge

The *unregister* operation affords no opportunity to provide or obtain unique knowledge.

5.4.4 Return Values Detail

The *unregister* operation *must* return a WsbdResult according to the following constraints.

5.4.4.1 Success

Status Value	`success`
Condition	The service accepted the unregistration request
Required Elements	`status` *must* be populated with the WsbdStatus literal `"success"`
Optional Elements	None

After unregistration, subsequent operations providing the unregistered session id *must* result in an `invalidId` status. An exception is the extremely unlikely case that the session id happens to be reused by the service during another registration.

As a consequence of idempotency, a session id does not need to ever have been registered successfully in order to *un*register successfully. Consequently, the *unregister* operation cannot return a status of `invalidId`.

5.4.4.2 Failure

Status Value	`failure`
Condition	The service could not unregister the session.
Required Elements	`status` *must* be populated with the literal `"failure"`
Optional Elements	`message` An informative description of the nature of the failure.

In practice, failure to unregister is expected to be a rare occurrence. Failure to unregister might occur if the service experiences a fault with an external system— such as a with centralized database used to track session registration and unregistration.

5.4.4.3 Sensor Busy

Status Value	`sensorBusy`
Condition	The service could not unregister the session because the biometric sensor is currently performing a sensor operation within the session being unregistered.
Required Elements	`status` *must* be populated with the literal `"sensorBusy"`
Optional Elements	None

This status *must only* be returned if (a) the sensor is busy and (b) the client making the request holds the lock (i.e., the session id provided matches that associated with the current service lock). Any client that does not hold the session lock *must not* result in a `sensorBusy` status.

EXAMPLE: The following sequence diagram illustrates a client that cannot unregister (Client A) and a client that can unregister (Client B), even though the service is performing a sensor operation for Client A.

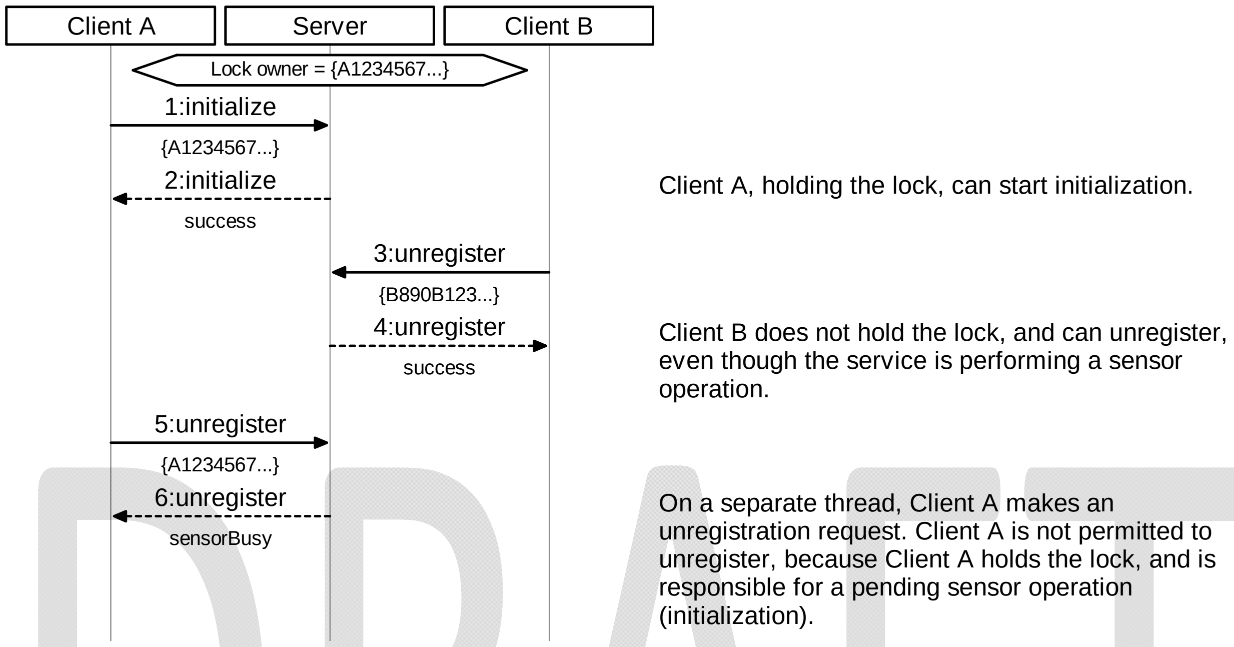
Client A
Server
Client B
Lock owner = {A1234567...}
1:initialize
{A1234567...}
2:initialize
success
Client A, holding the lock, can start initialization.
3:unregister
{B890B123...}
4:unregister
success
Client B does not hold the lock, and can unregister, even though the service is performing a sensor operation.
5:unregister
{A1234567...}
6:unregister
sensorBusy
On a separate thread, Client A makes an unregistration request. Client A is not permitted to unregister, because Client A holds the lock, and is responsible for a pending sensor operation (initialization).

Figure 4. Example of how an *unregister* operation can result in `sensorBusy`.

5.4.4.4 Bad Value

Status Value	`badValue`
Condition	The provided session id is not a well-formed UUID.

Required Elements	`status`
	must be populated with the literal `"badValue"`
	`badFields`
	must be populated with a WsbdStringArray that contains a single element, "`sessionId`". I.e. "`<element>sessionId</element>`". Note that `sessionId` is the literal "`sessionId`", not the ill-formed value.

Optional Elements	None

5.5 Try Lock

Description	Try to obtain the service lock
HTTP Method	`GET`
URL Template	`/lock/{sessionId}`
URL Parameters	`{sessionId}` (UUID, §4.1.1) Identity of the session requesting the service lock
Payload	Not applicable
Idempotent	Yes
Sensor Operation	No

5.5.1 WsbdResult Summary

`success`	`status="success"`
`failure`	`status="failure"` `message*=`informative message describing failure
`invalidId`	`status="invalidId"`
`lockHeldByAnother`	`status="lockHeldByAnother"`
`badValue`	`status="badValue"`, `badFields={"sessionId"}`

5.5.2 Usage Notes

The *try lock* operation attempts to obtain the service lock. The word "try" is used to indicate that the call returns immediately, and does not block until the lock is obtained. See §3.4.4 for detailed information about the WS-Biometric Devices concurrency and locking model.

5.5.3 Unique Knowledge

The *try lock* operation affords no opportunity to either provide or obtain unique knowledge.

5.5.4 Return Values Detail

The *try lock* operation *must* return a WsbdResult according to the following constraints.

5.5.4.1 Success

Status Value	`success`
Condition	The service was successfully locked to the provided session id.
Required Elements	`status` *must* be populated with the WsbdStatus literal `"success"`
Optional Elements	None

Clients that hold the service lock are permitted to perform sensor operations (§3.4.4). By idempotency (§3.4.6), if a client already holds the lock, subsequent *try lock* operations should also return `success`.

5.5.4.2 Failure

Status Value	`failure`
Condition	The service could not be locked to the provided session id.
Required Elements	`status` *must* be populated with the WsbdStatus literal `"failure"`
Optional Elements	`message` An informative description of the nature of the failure.

Services must reserve a `failure` status to report system or internal failures and prevent the acquisition of the lock. Most *try lock* operations that do not succeed will not produce a `failure` status, but more likely a `lockHeldByAnother` status (See §5.5.4.4 for an example).

5.5.4.3 Invalid Id

Status Value	`invalidId`
Condition	The provided session id is not registered with the service.
Required Elements	`status` *must* be populated with the WsbdStatus literal `"invalidId"`
Optional Elements	None

A session id is invalid if it does not correspond to an active registration. A session id may become unregistered from a service through explicit unregistration or, triggered automatically by the service due to inactivity (§5.4.4.1).

5.5.4.4 Lock Held by Another

Status Value	`lockHeldByAnother`
Condition	The service could not be locked to the provided session id because the lock is held by another client.
Required Elements	`status` *must* be populated with the WsbdStatus literal `"lockHeldByAnother"`
Optional Elements	None

EXAMPLE: The following sequence diagram illustrates a client that cannot obtain the lock (Client B) because it is held by another client (Client A).

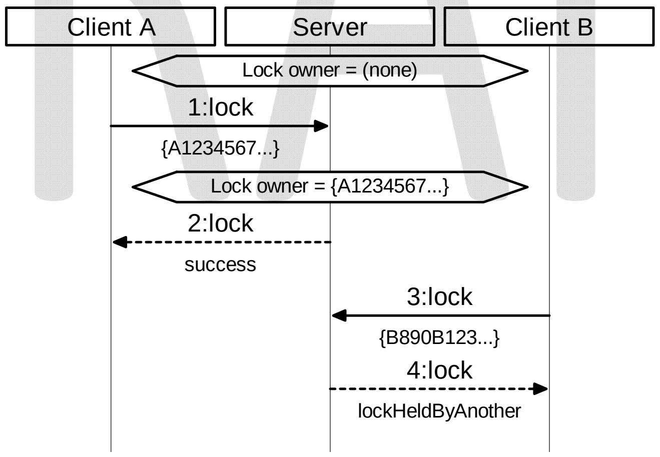

Figure 5. Example of a scenario yielding a `lockHeldByAnother` result.

5.5.4.5 Bad Value

Status Value	`badValue`
Condition	The provided session id is not a well-formed UUID.

Required Elements	`status`
	must be populated with the literal `"badValue"`
	`badFields`
	must be populated with a WsbdStringArray that contains a single element, "`sessionId`". I.e. "`<element>sessionId</element>`". Note that `sessionId` is the literal "`sessionId`", not the ill-formed value.

Optional Elements	None

5.6 Steal Lock

Description	Forcibly obtain the lock away from a peer client
HTTP Method	PUT
URL Template	`/lock/{sessionId}`
URL Parameters	`{sessionId}` (UUID, §4.1.1) Identity of the session requesting the service lock
Payload	Not applicable
Idempotent	Yes
Sensor Operation	No

5.6.1 WsbdResult Summary

`success`	`status="success"`
`failure`	`status="failure"` `message*=`informative message describing failure
`invalidId`	`status="invalidId"`
`badValue`	`status="badValue"`, `badFields={"sessionId"}`

5.6.2 Usage Notes

The *steal lock* operation allows a client to forcibly obtain the lock away from another client that already holds the lock. The purpose of this operation is to prevent a client that experiences a fatal error from forever preventing another client access to the service, and therefore, the biometric sensor.

5.6.2.1 Avoid Lock Stealing

System integrators *should* endeavor to reserve lock stealing for exceptional circumstances—such as when a fatal error prevents a client from releasing a lock. Lock stealing *should* not be used as the primary mechanism in which peer clients coordinate biometric sensor use.

5.6.2.2 Lock Stealing Prevention Period (LSPP)

To assist in coordinating access among clients, and to prevent excessive lock stealing, a service *may* trigger a time period that forbids lock stealing for each sensor operation. For convenience, this period of time will be referred to as the *lock stealing prevention period (LSPP).*

During the LSPP, all attempts to steal the service lock will fail. Consequently, if a client experiences a fatal failure during a sensor operation, then all peer clients need to wait until the service re-enables lock stealing.

All services *should* implement a non-zero LSPP to forbid locking. The recommended time for the LSPP is on the order of 100 seconds. Services that enforce an LSPP *must* start the LSPP at the start of each sensor operation request that proceeds far enough that a service would require sovereign sensor control. Examples of request that could proceed without sovereign sensor control are those that return a status of `invalidId`, or `badValue`. In these cases, there is no functional need to prevent lock stealing since the request does not directly involve the biometric sensor. *More specifics and examples might be necessary here. There may be a need to identify that different implementations may or may not trigger the LSPP on borderline status results, such as 'failure', and that each sensor operation itself might be a new decision point on whether or not to trigger a LSPP*

An LSPP ends after a fixed amount of time has elapsed, unless another sensor operation restarts the LSPP. *There should be a forward reference to how a service describes the lock-stealing period to a client. It may be desirable to have a (common info) name-value pair that reveals if the service is currently within an LSPP or not.*

Services should keep the length of the LSPP fixed throughout the service's lifecycle. It is recognized, however, that there may be use cases in which a variable LSPP timespan is desirable or required . Regardless, when determining the appropriate timespan, implements *should* carefully consider the tradeoffs between preventing excessive lock stealing, versus forcing all clients to wait until a service re-enables lock stealing.

5.6.2.3 Cancelation & (Lack of) Client Notification

Lock stealing *must* have no effect on any currently running sensor operations. It is possible that a client initiates a sensor operation, has its lock stolen away, yet the operation return successfully. *Subsequent* operations would yield a `lockNotHeld` status, which a client could use to indicate that their lock was stolen away from them. This cannot occur if the length of the LSPP is longer than any sensor operation. This will be the case for most service implementations, however, clients *should* accommodate for the possibility.

5.6.3 Unique Knowledge

The *steal lock* operation affords no opportunity to provide or obtain unique knowledge.

5.6.4 Return Values Detail

The *steal lock* operation *must* return a WsbdResult according to the following constraints.

5.6.4.1 Success

Status Value	`success`
Condition	The service was successfully locked to the provided session id.
Required Elements	`status` *must* be populated with the WsbdStatus literal "`success`"
Optional Elements	None

See the above usage notes for details (§3.4.4).

5.6.4.2 Failure

Status Value	`failure`
Condition	The service could not be locked to the provided session id.
Required Elements	`status` *must* be populated with the WsbdStatus literal "`failure`"
Optional Elements	`message` An informative description of the nature of the failure.

Most *steal lock* operations that yield a `failure` status will do so because the service receives a lock stealing request during a lock stealing prevention period (§5.6.2.2). Services *must* also reserve a `failure` status to other non-LSPP failures that prevent the acquisition of the lock.

Implementers *may* choose to use the optional `message` field to provide more information to an end-user as to the specific reasons for the failure. However (as with all other `failure` status results), clients *must* not depend on any particular content to make this distinction. Based on service metadata (*forward reference needed*) clients might infer if the failure was likely due to a LSPP failure or otherwise.

5.6.4.3 Invalid Id

Status Value	`invalidId`
Condition	The provided session id is not registered with the service.
Required Elements	`status` *must* be populated with the WsbdStatus literal `"invalidId"`
Optional Elements	None

A session id is invalid if it does not correspond to an active registration. A session id may become unregistered from a service through explicit unregistration or, triggered automatically by the service due to inactivity (§5.4.4.1).

5.6.4.4 Bad Value

Status Value	`badValue`
Condition	The provided session id is not a well-formed UUID.

Required Elements	`status`
	must be populated with the literal `"badValue"`
	`badFields`
	must be populated with a WsbdStringArray that contains a single element, “`sessionId`”. I.e. “`<element>sessionId</element>`”. Notice that `sessionId` is the literal “`sessionId`”, not the ill-formed value.

Optional Elements	None

5.7 Unlock

Description	Release the service lock
HTTP Method	PUT
URL Template	`/lock/{sessionId}`
URL Parameters	`{sessionId}` (UUID, §4.1.1) Identity of the session releasing the service lock
Payload	Not applicable
Idempotent	Yes
Sensor Operation	No

5.7.1 WsbdResult Summary

`success`	`status="success"`
`failure`	`status="failure"` `message*=`informative message describing failure
`invalidId`	`status="invalidId"`
`badValue`	`status="badValue"`, `badFields={"sessionId"}`

5.7.2 Usage Notes

The *unlock* operation release a service lock, making locking available to other clients. See §3.4.4 for detailed information about the WS-Biometric Devices concurrency and locking model.

5.7.2.1 Success

Status Value	`success`
Condition	The service returned to an unlocked state.
Required Elements	`status` *must* be populated with the WsbdStatus literal `"success"`
Optional Elements	None

Upon releasing the lock, a client is no longer permitted to perform any sensor operations (§3.4.4). By idempotency (§3.4.6), if a client already has released the lock, subsequent *try lock* operations should also return `success`.

5.8 Get Common Info

Description	Retrieve metadata about the service that does not depend on session-specific information, or sovereign control of the target biometric sensor
HTTP Method	`GET`
URL Template	`/info`
URL Parameters	None
Payload	Not applicable
Idempotent	Yes
Sensor Operation	No

5.8.1 WsbdResult Summary

`success`	`status="success"` `commonInfo=`dictionary containing service metadata
`failure`	`status="failure"` `message*=`informative message describing failure

5.8.2 Usage Notes

The *get common info* operation provides information about the service and target biometric sensor. This operation *must* return information that is both (a) independent of session, and (b) does not require the service to request dynamic and exclusive access to the target biometric sensor. The term “dynamic” is used to indicate that services *must not* control the target biometric sensor during a *get common info* operation itself. Implementers *may* (and are encouraged to) use service startup time to query the biometric sensor directly to create a cache of information and capabilities. The contents of this cache are a basis for *get common info* operations.

The *get common info* operation does *not* require that a client be registered with the service, and therefore, unlike other operations, take a session id as a URL parameter.

5.8.3 Unique Knowledge

The *get common info* operation affords ample opportunity for a service to provide unique component knowledge to a client. A series of well-accepted name-value pairs that could provide a baseline of functionality across all services is defined in detail in Appendix A.

5.8.4 Return Values Detail

The *steal lock* operation *must* return a WsbdResult according to the following constraints.

5.8.4.1 Success

Status Value	`success`
Condition	The service provides service metadata

Required Elements	`status`
	must be populated with the WsbdStatus literal `"success"`
	`commonInfo`
	must be populated with a WsbdResult that contains service metadata

Optional Elements	None

See Appendix 0 for detailed information on common metadata.

5.8.4.2 Failure

Status Value	`failure`
Condition	The service cannot provide service metadata
Required Elements	`status` *must* be populated with the WsbdStatus literal `"failure"`
Optional Elements	`message` An informative description of the nature of the failure.

5.9 Get Detailed Info

Description	Retrieve metadata about the service that may depend on either session-specific information or sovereign control of the target biometric sensor
HTTP Method	`GET`
URL Template	`/info/{sessionId}`
URL Parameters	`{sessionId}` (UUID, §4.1.1) Identity of the session requesting detailed metadata
Payload	Not applicable
Idempotent	Yes
Sensor Operation	Yes

5.9.1 WsbdResult Summary

`success`	`status="success"` `detailedInfo=`dictionary containing service metadata
`failure`	`status="failure"` `message*=`informative message describing failure
`invalidId`	`status="invalidId"`
`canceled`	`status="canceled"`
`canceledWithSensorFailure`	`status="canceledWithSensorFailure"`
`sensorFailure`	`status="sensorFailure"`
`lockNotHeld`	`status="lockNotHeld"`
`lockHeldByAnother`	`status="canceled"`
`sensorNeedsInitialization`	`status="canceledWithSensorFailure"`
`sensorBusy`	`status="sensorBusy"`
`sensorTimeout`	`status="sensorTimeout"`
`badValue`	`status="badValue"`, `badFields={"sessionId"}`

5.9.2 Usage Notes

The *get detailed info* operation provides information about the service and target biometric sensor. Unlike its counterpart (*get common info)*, *get detailed info* *may* return information that is (a) dependent of session and/or (b) requires dynamic and exclusive access to the target biometric sensor. That is, a *get detailed info* *may* directly communicate with the target biometric sensor within a *get detailed info* operation. Therefore, the in order to be successful, a *get detailed info* operation requires that the client hold the service lock.

5.9.3 Unique Knowledge

The *get detailed info* operation affords ample opportunity for a service to provide unique component knowledge to a client. A series of well-accepted name-value pairs that could provide a baseline of functionality across all services is defined in detail in Appendix B.

5.9.4 Return Values Detail

The *get detailed info* operation *must* return a WsbdResult according to the following constraints.

5.9.4.1 Success

Status Value	`success`
Condition	The service provides service metadata
Required Elements	`status` *must* be populated with the WsbdStatus literal `"success"`

	`commonInfo` *must* be populated with a WsbdResult that contains detailed service metadata
Optional Elements	None

See Appendix B for full information on detailed metadata.

5.9.4.2 Failure

Status Value	`failure`
Condition	The service cannot provide service metadata due to service (not target biometric sensor) error.
Required Elements	`status` *must* be populated with the WsbdStatus literal "`failure`"
Optional Elements	`message` An informative description of the nature of the failure.

Services *must* only use this status to report failures that occur within the web service, not the target biometric sensor (see §5.9.4.4, §5.9.4.5).

5.9.4.3 Invalid Id

Status Value	`invalidId`
Condition	The provided session id is not registered with the service.
Required Elements	`status` *must* be populated with the WsbdStatus literal "`invalidId`"
Optional Elements	None

A session id is invalid if it does not correspond to an active registration. A session id may become unregistered from a service through explicit unregistration or, triggered automatically by the service due to inactivity (§5.4.4.1).

5.9.4.4 Canceled

Status Value	`canceled`
Condition	The operation was interrupted by a cancelation request.
Required Elements	`status` *must* be populated with the WsbdStatus literal "`canceled`"
Optional Elements	None

Like all sensor operations, the *get detailed info* operation can be explicitly canceled by the originating a client, a client that stole the service lock, or, automatically by the service if the service determines that the operation has been running for too long (§*forward reference needed*).

5.9.4.5 Canceled with Sensor Failure

Status Value	`canceledWithSensorFailure`
Condition	The operation was interrupted by a cancelation request during which the target biometric sensor experienced a failure
Required Elements	`status` *must* be populated with the WsbdStatus literal "`canceledWithSensorFailure`"
Optional Elements	`message` An informative description of the nature of the failure.

Services *must* return a `canceledWithSensorFailure` result if a cancelation request caused a failure within the target biometric sensor. Clients receiving this result may need to perform initialization to restore full functionality.

5.9.4.6 Sensor Failure

Status Value	`sensorFailure`
Condition	The service could not retrieve detailed information due to a failure within the target biometric sensor.
Required Elements	`status` *must* be populated with the WsbdStatus literal `"sensorFailure"`
Optional Elements	`message` An informative description of the nature of the failure.

A `sensorFailure` status *must* only be used to report failures that occurred within the target biometric sensor, not a failure within the web service (§5.10.4.2).

5.9.4.7 Lock Not Held

Status Value	`lockNotHeld`
Condition	The service could not retrieve detailed information because the requesting client does not hold the lock.
Required Elements	`status` *must* be populated with the WsbdStatus literal `"lockNotHeld"`
Optional Elements	None

Initialization is a sensor operation, and requires that the requesting client holds the service lock.

5.9.4.8 Lock Held by Another

Status Value	`lockHeldByAnother`
Condition	The service could not retrieve detailed information because the lock is held by another client.
Required Elements	`status` *must* be populated with the WsbdStatus literal `"lockHeldByAnother"`
Optional Elements	None

5.9.4.9 Sensor Needs Initialization

Status Value	`sensorNeedsInitialization`
Condition	The service could not retrieve detailed information because the target biometric sensor has not been initialized.
Required Elements	`status` *must* be populated with the WsbdStatus literal `"sensorNeedsInitialization"`
Optional Elements	None

Services *should* be able to provide detailed sensor metadata without initialization; however, this is not strictly necessary. Similarly, clients *should* assume that retrieving sensor metadata requires initialization

5.9.4.10 Sensor Busy

Status Value	`sensorBusy`
Condition	The service could not retrieve detailed information because the service is already performing a different sensor operation for the requesting client.
Required Elements	`status` *must* be populated with the WsbdStatus literal `"sensorBusy"`
Optional Elements	None

5.9.4.11 Sensor Timeout

Status Value	`sensorTimeout`
Condition	The service could not retrieve detailed information because the target biometric sensor took too long to complete the request.

Required Elements	`status` *must* be populated with the WsbdStatus literal `"sensorTimeout"`
Optional Elements	None

A service did not receive a timely response from the target biometric sensor. Note that this condition is distinct from the client's originating HTTP request, which may have its own, independent timeout. *A forward reference to timeout metadata may be needed.*

5.9.4.12 Bad Value

Status Value	`badValue`
Condition	The provided session id is not a well-formed UUID.

Required Elements	`status`
	must be populated with the literal `"badValue"`
	`badFields`
	must be populated with a WsbdStringArray that contains a single element, "`sessionId`". I.e. "`<element>sessionId</element>`". Notice that `sessionId` is the literal "`sessionId`", not the ill-formed value.

Optional Elements	None

5.10 Initialize

Description	Initialize the target biometric sensor
HTTP Method	`GET`
URL Template	`/initialize/{sessionId}`
URL Parameters	`{sessionId}` (UUID, §4.1.1) Identity of the session requesting detailed metadata
Payload	Not applicable
Idempotent	Yes
Sensor Operation	Yes

5.10.1 WsbdResult Summary

`success`	`status="success"`
`failure`	`status="failure"` `message*`=informative message describing failure
`invalidId`	`status="invalidId"`
`canceled`	`status="canceled"`
`canceledWithSensorFailure`	`status="canceledWithSensorFailure"`
`sensorFailure`	`status="sensorFailure"`
`lockNotHeld`	`status="lockNotHeld"`
`lockHeldByAnother`	`status="canceled"`
`sensorBusy`	`status="sensorBusy"`
`sensorTimeout`	`status="sensorTimeout"`
`badValue`	`status="badValue"`, `badFields={"sessionId"}`

5.10.2 Usage Notes

The *initialize* operation prepares the target biometric sensor for (other) sensor operations.

Although not strictly necessary, services *should* directly map this operation to the initialization of the target biometric sensor, unless the service can otherwise determine that the target biometric sensor is in a fully operation state. This would enable the ability of a client to attempt a manual reset of a sensor that has entered a faulty state—particularly useful in physically separated service implementations.

Some biometric sensors have requirement for explicit initialization. Services exposing such a sensor *should* immediately return a `success` result.

5.10.3 Unique Knowledge

The *initialize* operation affords no opportunity to provide or obtain unique knowledge.

5.10.4 Return Values Detail

5.10.4.1 Success

Status Value	`success`
Condition	The service successfully initialized the target biometric sensor
Required Elements	`status` *must* be populated with the WsbdStatus literal `"success"`
Optional Elements	None

5.10.4.2 Failure

Status Value	`failure`
Condition	The service experienced a fault that prevented successful initilazation.
Required Elements	`status` *must* be populated with the WsbdStatus literal "`failure`"
Optional Elements	`message` An informative description of the nature of the failure.

A `failure` status *must* only be used to report failures that occurred within the web service, not within the target biometric sensor (§5.10.4.5, §5.10.4.6)

5.10.4.3 Invalid Id

Status Value	`invalidId`
Condition	The provided session id is not registered with the service.
Required Elements	`status` *must* be populated with the WsbdStatus literal "`invalidId`"
Optional Elements	None

A session id is invalid if it does not correspond to an active registration. A session id may become unregistered from a service through explicit unregistration or, triggered automatically by the service due to inactivity (§5.4.4.1).

5.10.4.4 Canceled

Status Value	`canceled`
Condition	The initialization operation was interrupted by a cancelation request.
Required Elements	`status` *must* be populated with the WsbdStatus literal "`canceled`"
Optional Elements	None

The *initialize* operation can be explicitly canceled by the originating a client, a client that stole the service lock, or, automatically by the service if the service determines that the operation has been running for too long (§*forward reference needed*).

5.10.4.5 Canceled with Sensor Failure

Status Value	`canceledWithSensorFailure`
Condition	The initialization operation was interrupted by a cancelation request and the target biometric sensor experienced a failure
Required Elements	`status` *must* be populated with the WsbdStatus literal "`canceledWithSensorFailure`"
Optional Elements	`message` An informative description of the nature of the failure.

Services *must* return a `canceledWithSensorFailure` result if a cancelation request caused a failure within the target biometric sensor. Clients receiving this result may need to reattempt the initialization request to restore full functionality.

5.10.4.6 Sensor Failure

Status Value	`sensorFailure`
Condition	The initialization failed due to a failure within the target biometric sensor
Required Elements	`status` *must* be populated with the WsbdStatus literal "`sensorFailure`"
Optional Elements	`message` An informative description of the nature of the failure.

A `sensorFailure` status *must* only be used to report failures that occurred within the target biometric sensor, not a failure within the web service (§5.10.4.2).

5.10.4.7 Lock Not Held

Status Value	`lockNotHeld`
Condition	Initialization could not be performed because the requesting client does not hold the lock
Required Elements	`status` *must* be populated with the WsbdStatus literal `"lockNotHeld"`
Optional Elements	None

Initialization is a sensor operation, and requires that the requesting client holds the service lock.

5.10.4.8 Lock Held by Another

Status Value	`lockHeldByAnother`
Condition	Initialization could not be performed because the lock is held by another client.
Required Elements	`status` *must* be populated with the WsbdStatus literal `"lockHeldByAnother"`
Optional Elements	None

5.10.4.9 Sensor Busy

Status Value	`sensorBusy`
Condition	Initialization could not be performed because the service is already performing a different sensor operation for the requesting client.
Required Elements	`status` *must* be populated with the WsbdStatus literal `"sensorBusy"`
Optional Elements	None

5.10.4.10 Sensor Timeout

Status Value	`sensorTimeout`
Condition	Initialization could not be performed because the target biometric sensor took too long to complete the initialization request.
Required Elements	`status` *must* be populated with the WsbdStatus literal `"sensorTimeout"`
Optional Elements	None

A service did not receive a timely response from the target biometric sensor. Note that this condition is distinct from the client's originating HTTP request, which may have its own, independent timeout. *A forward reference to timeout metadata may be needed.*

5.11 Get Configuration

Description	Get the target biometric sensor's current configuration
HTTP Method	`GET`
URL Template	`/configure/{sessionId}`
URL Parameters	`{sessionId}` (UUID, §4.1.1) Identity of the session requesting the configuration
Payload	Not applicable
Idempotent	Yes
Sensor Operation	Yes

5.11.1 WsbdResult Summary

`success`	`status="success"` `configuration=`current configuration of the sensor (WsbdResult, §4.2)
`failure`	`status="failure"` `message*=`informative message describing failure
`invalidId`	`status="invalidId"`
`canceled`	`status="canceled"`
`canceledWithSensorFailure`	`status="canceledWithSensorFailure"`
`sensorFailure`	`status="sensorFailure"`
`lockNotHeld`	`status="lockNotHeld"`
`lockHeldByAnother`	`status="canceled"`
`sensorNeedsInitialization`	`status="sensorNeedsInitialization"`
`sensorNeedsConfiguration`	`status="sensorNeedsConfiguration"`
`sensorBusy`	`status="sensorBusy"`
`sensorTimeout`	`status="sensorTimeout"`
`badValue`	`status="badValue"`, `badFields={"sessionId"}`

5.11.2 Usage Notes

The *get configuration* operation retrieves the service's current configuration.

5.11.3 Unique Knowledge

The *get configuration* operation affords ample opportunity for a service to provide unique component knowledge to a client. A series of well-accepted name-value pairs that could provide a baseline of functionality across all services is defined in detail in Appendix C.

5.11.4 Return Values Detail

The *get configuration* operation *must* return a WsbdResult according to the following constraints.

5.11.4.1 Success

Status Value	`success`
Condition	The service provides the current configuration

Required Elements	`status`
	must be populated with the WsbdStatus literal `"success"`
	`configuration`
	must be populated with a WsbdResult that contains the service's current configuration

Optional Elements	None

See Appendix C for detailed information regarding configuration.

5.11.4.2 Failure

Status Value	`failure`
Condition	The service cannot provide the current configuration due to service (not target biometric sensor) error.
Required Elements	`status` *must* be populated with the WsbdStatus literal "`failure`"
Optional Elements	`message` An informative description of the nature of the failure.

Services *must* only use this status to report failures that occur within the web service, not the target biometric sensor (see §5.11.4.5, §5.11.4.6).

5.11.4.3 Invalid Id

Status Value	`invalidId`
Condition	The provided session id is not registered with the service.
Required Elements	`status` *must* be populated with the WsbdStatus literal "`invalidId`"
Optional Elements	None

A session id is invalid if it does not correspond to an active registration. A session id may become unregistered from a service through explicit unregistration or, triggered automatically by the service due to inactivity (§5.4.4.1).

5.11.4.4 Canceled

Status Value	`canceled`
Condition	The *get configuration* operation was interrupted by a cancelation request.
Required Elements	`status` *must* be populated with the WsbdStatus literal "`canceled`"
Optional Elements	None

Like all sensor operations, the *get configuration* operation can be explicitly canceled by the originating a client, a client that stole the service lock, or, automatically by the service if the service determines that the operation has been running for too long (§A.1.1).

5.11.4.5 Canceled with Sensor Failure

Status Value	`canceledWithSensorFailure`
Condition	The *get configuration* operation was interrupted by a cancelation request during which the target biometric sensor experienced a failure
Required Elements	`status` *must* be populated with the WsbdStatus literal "`canceledWithSensorFailure`"
Optional Elements	`message` An informative description of the nature of the failure.

Services *must* return a `canceledWithSensorFailure` result if a cancelation request caused a failure within the target biometric sensor. Clients receiving this result may need to perform initialization to restore full functionality.

5.11.4.6 Sensor Failure

Status Value	`sensorFailure`
Condition	The configuration could not be queried due to a failure within the target biometric sensor.

Required Elements	`status` *must* be populated with the WsbdStatus literal `"sensorFailure"`
Optional Elements	`message` An informative description of the nature of the failure.

A `sensorFailure` status *must* only be used to report failures that occurred within the target biometric sensor, not a failure within the web service (§5.10.4.2).

5.11.4.7 Lock Not Held

Status Value	`lockNotHeld`
Condition	The configuration could not be queried because the requesting client does not hold the lock.
Required Elements	`status` *must* be populated with the WsbdStatus literal `"lockNotHeld"`
Optional Elements	None

Initialization is a sensor operation, and requires that the requesting client holds the service lock.

5.11.4.8 Lock Held by Another

Status Value	`lockHeldByAnother`
Condition	The configuration could not be queried because the lock is held by another client.
Required Elements	`status` *must* be populated with the WsbdStatus literal `"lockHeldByAnother"`
Optional Elements	None

5.11.4.9 Sensor Needs Initialization

Status Value	`sensorNeedsInitialization`
Condition	The configuration could not be queried because the target biometric sensor has not been initialized.
Required Elements	`status` *must* be populated with the WsbdStatus literal `"sensorNeedsInitialization"`
Optional Elements	None

Services *should* be able to provide the sensors configuration without initialization; however, this is not strictly necessary. Similarly, clients *should* assume that configuration will require initialization.

5.11.4.10 Sensor Needs Configuration

Status Value	`sensorNeedsConfiguration`
Condition	The configuration could not be queried because the target biometric sensor has not been initialized.
Required Elements	`status` *must* be populated with the WsbdStatus literal `"sensorNeedsConfiguration"`
Optional Elements	None

Services *may* require configuration to be set before a configuration can be retrieved if a service does not provide a valid default configuration.

5.11.4.11 Sensor Busy

Status Value	`sensorBusy`
Condition	The configuration could not be queried because the service is already performing a different sensor operation for the requesting client.
Required Elements	`status` *must* be populated with the WsbdStatus literal `"sensorBusy"`
Optional Elements	None

5.11.4.12 Sensor Timeout

Status Value	`sensorTimeout`
Condition	The configuration could not be queried because the target biometric sensor took too long to complete the request.
Required Elements	`status` *must* be populated with the WsbdStatus literal `"sensorTimeout"`
Optional Elements	None

A service did not receive a timely response from the target biometric sensor. Note that this condition is distinct from the client's originating HTTP request, which may have its own, independent timeout. *A forward reference to timeout metadata may be needed.*

5.11.4.13 Bad Value

Status Value	`badValue`
Condition	The provided session id is not a well-formed UUID.

Required Elements	`status`
	must be populated with the literal `"badValue"`
	`badFields`
	must be populated with a WsbdStringArray that contains a single element, "`sessionId`". I.e. "`<element>sessionId</element>`". Notice that `sessionId` is the literal "`sessionId`", not the ill-formed value.

Optional Elements	None

5.12 Set Configuration

Description	Set the target biometric sensor's configuration
HTTP Method	`POST`
URL Template	`/configure/{sessionId}`
URL Parameters	`{sessionId}` (UUID, §4.1.1) Identity of the session requesting the configuration
Payload	Desired sensor configuration (WsbdDictionary, §4.1.2)
Idempotent	Yes
Sensor Operation	Yes

5.12.1 WsbdResult Summary

`success`	`status="success"`
`failure`	`status="failure"` `message*=`informative message describing failure
`invalidId`	`status="invalidId"`
`canceled`	`status="canceled"`
`canceledWithSensorFailure`	`status="canceledWithSensorFailure"`
`sensorFailure`	`status="sensorFailure"`
`lockNotHeld`	`status="lockNotHeld"`
`lockHeldByAnother`	`status="canceled"`
`sensorNeedsInitialization`	`status="sensorNeedsInitialization"`
`sensorBusy`	`status="sensorBusy"`
`sensorTimeout`	`status="sensorTimeout"`
`unsupported`	`status="unsupported"`, `badFields={` set of field names `}`

`badValue`	`status="badValue"`, `badFields={"sessionId"}`
	or
	`status="badValue"`, `badFields={` set of field names `}`

`noSuchParameter`	`status="unsupported"`, `badFields={` set of field names `}`

5.12.2 Usage Notes

The *set configuration* operation sets the configuration of a service's target biometric sensor.

5.12.3 Unique Knowledge

The *set configuration* operation affords ample opportunity for a service to provide unique component knowledge to a client. A series of well-accepted name-value pairs that could provide a baseline of functionality across all services is defined in detail in Appendix C.

5.12.4 Usage Notes

5.12.4.1 Payload

The *set configuration* operation is the only operation that takes input within the body of the HTTP request. The desired configuration *must* be sent as a single, unnamed WsbdDictionary.

EXAMPLE: The following represents a ‘raw’ request to configure a service at `http://10.0.0.2:7000/Sensor` such that `string1=value` and `integer1=1`. In this example, each `value` element contains fully qualified namespace information, although this is not necessary.

```
POST http://10.0.0.2:7000/Sensor/configure/1678e0fa-b578-4234-bb59-6f2f92d7b80c HTTP/1.1
Content-Type: application/xml
Host: 10.0.0.2:7000
Content-Length: 351
Expect: 100-continue

<WsbdDictionary xmlns:i="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://itl.nist.gov/w
sbd/L1"><item><key>string1</key><value xmlns:d3p1="http://www.w3.org/2001/XMLSchema" i:type="d3p
1:string">value</value></item><item><key>integer1</key><value xmlns:d3p1="http://www.w3.org/2001
/XMLSchema" i:type="d3p1:int">1</value></item></WsbdDictionary>
```

This example should be updated to use the correct namespace

5.13 Capture

5.14 Get Content Type

5.15 Download

5.16 Thrifty Download

5.17 Cancel

Description	Cancel the current sensor operation
HTTP Method	`POST`
URL Template	`/cancel/{sessionId}`
URL Parameters	`{sessionId}` (UUID, §4.1.1) Identity of the session requesting cancelation
Payload	Not applicable
Idempotent	Yes
Sensor Operation	Yes

5.17.1 WsbdResult Summary

`success`	`status="success"`
`failure`	`status="failure"` `message*`=informative message describing failure
`invalidId`	`status="invalidId"`
`lockNotHeld`	`status="lockNotHeld"`
`lockHeldByAnother`	`status="lockHeldByAnother"`
`badValue`	`status="badValue"`, `badFields={"sessionId"}`

5.17.2 Usage Notes

The *cancel* operation stops any currently running sensor operation; it has no effect on non-sensor operations. If cancelation of an active sensor operation is successful, *cancel* operation receives `success` result, while the canceled operation receives a `canceled` (or `canceledWithSensorFailure`) result. As long as the operation is canceled, the *cancel* operation itself receives a `success` result, regardless if cancelation caused a sensor failure. In other words, if cancelation caused a fault within the target biometric sensor, as long as the sensor operation has stopped running, the *cancel* operation is considered to be successful.

Clients are responsible for canceling all non-sensor operations via client-side mechanisms only. Cancellation of sensor operations requires a separate service operation, since a service may need to “manually” interrupt a busy sensor. A service that had its client terminate an operation would have no way to easily determine that a cancellation request.

A Common Info

This contains detailed information on the set of "well-accepted" metadata potentially available from a *get common info* operation.

A.1 Data Dictionary

A.2 Parameter Details

A.1.1 Connections

The following parameters describe how the service handles session lifetimes and registrations.

A.1.1.1 Last Updated

Formal Name	`lastUpdated`
Data Type	`xsd:dateTime`
Required	Yes

This parameter provides a timestamp of when the service last *updated* the (other) parameters provided in the common info dictionary. The timestamp *must* include time zone information. Implementers *should* expect clients to use this timestamp to detect if any cached values of the (other) common info parameters may have changed.

A.1.1.2 Inactivity Timeout

Formal Name	`inactivityTimeout`
Data Type	`xsd:nonNegativeInteger`
Required	Yes

This parameter describes how long, in seconds, a session may be inactive before it may be automatically closed by the service. A value of '0' indicates that the service never drops sessions due to inactivity.

The inactivity of a session is measured as the time elapsed between the time at which last operation made with the session's id and the current time. Services *must* only use the session id to determine a session's inactivity time. For example, a service does not maintain different inactivity timeouts for requests that use the same session id, but originate from two different IP addresses.

A.2.1.1 Maximum Concurrent Sessions

Formal Name	`maximumConcurrentSessions`
Data Type	`xsd:positiveInteger`
Required	Yes

This parameter describes the maximum number of concurrent sessions a service can maintain. Upon startup, a service *must* have zero concurrent sessions. When a client registers successfully (§5.3), the service increases its count of concurrent sessions by one. Upon successful unregistration (§5.4), the service decreases its count of concurrent sessions by one.

A.1.1.3 Least Recently Used (LRU) Sessions Automatically Dropped

Formal Name	`autodropLruSessions`
Data Type	`xsd:boolean`
Required	Yes

This parameter describes whether or not the service automatically unregisters the least-recently-used session when the service has reached its maximum number of concurrent session. If *true*, then upon receiving a registration request, the service may drop the least-recently used session if the maximum number of concurrent sessions has already been reached. If *false*, then any registration request that would cause the service to exceed its maximum number of concurrent sessions results in failure.

A.2.2 Timeouts

Clients *should* be written in such a manner that a sensor operation does not block indefinitely. However, since different services may offer a variety of sensors and capabilities, clients require a means to determine appropriate timeouts. The timeouts in this subsection are

Note that these timeouts do not include any round-trip and network delay—clients *should* add an additional window to accommodate

A.2.2.1 Initialization Timeout

A.2.2.2 Detailed Info Timeout

Formal Name	`initializationTimeout`
Data Type	`xsd:nonNegativeInteger`
Required	Yes

A.2.2.3 Get Configuration Timeout

A.2.2.4 Set Configuration Timeout

A.2.2.5 Capture Timeout

B Detailed Info

This section will contain detailed information on the set of “well-accepted” detailed metadata potentially available from a detailed info operation.

C Configuration

This section will contain detailed information on the set of "well-accepted" configurations gettable and settable via configuration operations.

D Download Data

D.1 MIME Type

D.2 Thrifty Download

E Security Profiles

E.1 None

This section will describe a configuration in which all security is established at a lower network level. Implementors *should* only use this security profile for testing and development.

E.2 HTTPS

This section will describe a configuration in which HTTPS is used for encryption purposes. There should be a note about the optional use of

- *Basic or digest authentication*
- *Client-side certificates*

E.3 OpenID & OAuth

This section might describe a configuration using OpenID and/or OAuth to control access to the service at a fine level of granularity. This may be out of scope for this revision of the document.

F Conformance

A service can claim **Level 0** conformance to this specification if the service conforms to all of this specification with exception of the metadata appendixes.

Conform "as a biometric sensor" vs "web cam" vs "fingerprint scanner"

G Pending Issues

- Currently, it is assumed that messages are returned from the service in a single language. Integrated multilingual support might be supported by a special sensor service configuration operation, or, by supporting multiple languages within the messages returned by a service.
- *Should we force every service to provide a fully functional, default configuration? If so, then we may not need a configurationNeeded status.*

H Acknowledgments

The authors thank the following individuals for their participation in the creation of this specification.

Bert Coursey, Department of Homeland Security, Science & Technology Directorate
Michael Garris, National Institute of Standards and Technology, Information Technology Lab
John Manzo, Federal Bureau of Investigation
Scott Swann, Federal Bureau of Investigation
Cathy Tilton, Daon Inc.
Bradford Wing, National Institute of Standards and Technology, Information Technology Lab

I Revision History

Draft 0—Initial release. Operations and data types are well defined, but detailed documentation is not yet complete. Appendixes (metadata, conformance, and security profiles) are not yet written.