TCG Trusted Network Communications
TNC IF-IMV

Specification Version 1.4
Revision 11
5 December 2014
Published

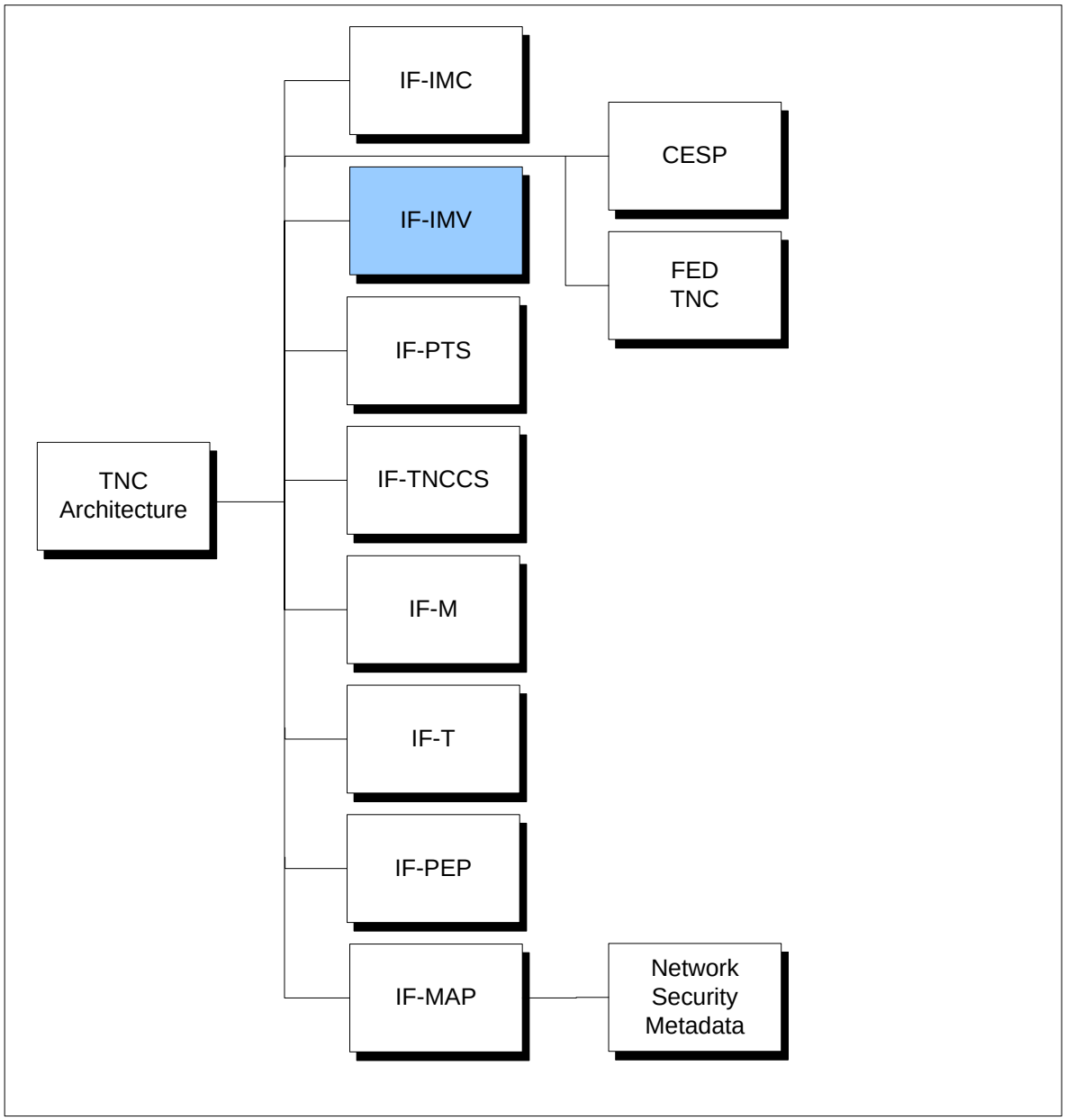
Contact:
admin@trustedcomputinggroup.org

TCG

TCG PUBLISHED

Disclaimers, Notices, and License Terms

TNC Document Roadmap



Acknowledgements

The TCG wishes to thank all those who contributed to this specification. This document builds on considerable work done in the various working groups in the TCG.

Special thanks to the members of the TNC contributing to this document:

Aman Garg	3Com
Bipin Mistry	3Com
Mahalingam Mani	Avaya
Padma Krishnaswamy	Battelle Memorial Institute
Eric Fleischman	Boeing
Richard Hill	Boeing
Steven Venema	Boeing
Nancy Cam-Winget	Cisco Systems
Scott Pope	Cisco Systems
Max Pritikin	Cisco Systems
Allan Thomson	Cisco Systems
Mark Beadles (Editor of IF-IMV 1.0)	Endforce, Inc.
Nicolai Kuntze	Fraunhofer Institute for Secure Information Technology (SIT)
Hidenobu Ito	Fujitsu Limited
Sung Lee	Fujitsu Limited
Kazuaki Nimura	Fujitsu Limited
Boris Balacheff	Hewlett-Packard
Paul Crandell	Hewlett-Packard
Mauricio Sanchez	Hewlett-Packard
Ira McDonald	High North
Dr. Josef von Helden	Hochschule Hannover
Dr. Andreas Steffen	HSR University of Applied Sciences Rapperswil
Diana Arroyo (Editor of IF-IMV 1.2 and 1.3)	IBM
Lee Terrell	IBM
Frank Yeh	IBM
Stuart Bailey	Infoblox
James Tan	Infoblox
Tina Bird	InfoExpress, Inc.
Ravi Sahita	Intel Corporation
Ned Smith	Intel Corporation
Barbara Nelson	iPass
Chris Trytten	iPass
Steve Hanna (Editor, TNC-WG co-chair)	Juniper Networks
Clifford Kahn	Juniper Networks
Lisa Lorenzin	Juniper Networks
Hirendra Rathor (Editor of IF-IMV 1.3)	Juniper Networks
John Jerrim	Lancope, Inc.
Gene Chang	Meetinghouse Data Communications
Alex Romanyuk	Meetinghouse Data Communications
John Vollbrecht	Meetinghouse Data Communications
Atul Shah (TNC-WG Co-Chair)	Microsoft
Jon Baker	MITRE
Charles Schmidt	MITRE
Sandilya Garimella	Motorola
Rainer Enders	NCP Engineering

Joseph Tardo	Nevis Networks
Pasi Eronen	Nokia Corporation
Meenakshi Kaushik	Nortel Networks
Dick Wilkins	Phoenix Technologies
Thomas Hardjono	SignaCert, Inc.
Carolin Latze	Swisscom
Babak Salimi	Sygate Technologies, Inc.
Bryan Kingsford	Symantec
Paul Sangster	Symantec
Richard Struse	U.S. Department of Homeland Security
Mike Boyle	U.S. National Security Agency
Emily Doll	U.S. National Security Agency
Jessica Fitzgerald-McKay	U.S. National Security Agency
Mary Lessels	U.S. National Security Agency
Chris Salter	U.S. National Security Agency
Jeff Six	U.S. National Security Agency
Rod Murchison	Vernier Networks
Michele Sommerstad	Vernier Networks
Scott Cochrane	Wave Systems
Greg Kazmierczak	Wave Systems

Table of Contents

1 Scope and Audience

The Trusted Network Communications Work Group (TNC-WG) has defined an open solution architecture that enables network operators to enforce policies regarding the security state of endpoints in order to determine whether to grant access to a requested network infrastructure. This security assessment of each endpoint is performed using a set of asserted integrity measurements covering aspects of the operational environment of the endpoint. These measurements are obtained from Integrity Measurement Collectors, which may base the measurements on Trusted Platform Module (TPM)-based measurements or software observations alone. Part of the TNC architecture is IF-IMV, a standard interface between Integrity Measurement Verifiers and the TNC Server. This document defines and specifies IF-IMV.

Architects, designers, developers and technologists who wish to implement, use, or understand IF-IMV should read this document carefully. Before reading this document any further, the reader should review and understand the TNC architecture as described in [1].

2 Purpose and Requirements

2.1 Purpose of IF-IMV

This document describes and specifies IF-IMV, a critical interface in the Trusted Computing Group's Trusted Network Communications (TNC) architecture. IF-IMV is the interface between Integrity Measurement Verifiers (IMVs) and a TNC Server (TNCS). It is closely related to IF-IMC [9], the interface between Integrity Measurement Clients (IMCs) and a TNC Client (TNCC).

IF-IMV is primarily used to receive integrity measurements sent from client-side Integrity Measurement Collectors (IMCs) to corresponding Integrity Measurement Verifiers (IMVs) and to enable message exchanges between the IMCs and the IMVs. These message exchanges occur within Integrity Check Handshakes, each of which is an example of a TCG attestation protocol in the context of the TNC architecture. It also allows IMVs to supply their recommendations to the TNCS.

An API-based approach has been chosen as the preferred embodiment of IF-IMV, similar to IF-IMC [9]. See Section 3 of this document for description of the abstract API and Section 4 for specific platform bindings. There is also a protocol binding for IF-IMV [23].

2.2 Summary of Changes since IF-IMV 1.3

The following changes have been made to IF-IMV since the last version (IF-IMV 1.3):

- Added standard way for TNCSs to expose information about Access Requester identity to IMVs.

2.3 Supported Use Cases

Use cases that this version of IF-IMV supports are as follows:

- An IMV and TNCS that support the same platform binding are installed on an endpoint. The TNCS finds and loads the IMV. Then it runs one or more Integrity Check Handshakes. The IMV and TNCS may use any of the features of IF-IMV.
- A TNCS has restricted privileges. It loads IMVs and runs one or more Integrity Check Handshakes.
- A TNCS that supports the Java Platform Binding runs with generous privileges but chooses to run IMVs with restricted privileges for security reasons. It loads IMVs and runs one or more Integrity Check Handshakes.
- An IMV and a TNCS both support the reason string extensions to IF-IMV. An IMV provides a reason string to a TNCS, giving the reason for its IMV Action Recommendation. The TNCS logs this reason and/or passes it on to the TNCC through IF-TNCCS 2.0 or another protocol. At the TNCC, the reason information may be displayed to the user (perhaps in a detailed view). The reason string may be in the endpoint user's preferred language or (if that language is not available) in another language.
- A TNCS is running. When an IMV is installed or uninstalled, the TNCC notices this and loads or unloads the IMV.
- A TNCS that provides information to IMVs that can be used to adapt their behavior to accomodate one or more of the the following cases:
    - A TNCS that provides IMVs information about a limited number of round trips and/or a small amount of data for IMC-IMV messages. IMVs can use this information to adapt their messaging to accommodate these limitations.

  - A TNCS that supports Diffie-Hellman Pre-Negotiation (as described in IF-T for Tunneled EAP Methods) which provides a Unique-Value-1 to the PTS-IMV, so that this value can be used for verifying the PTS-IMC's Integrity Report.

  - A TNCS that supports TLS-Unique (as described in PT-TLS [22]) provides the TLS-Unique value to the PTS-IMV or other IMVs, so that this value can be used in the TPM_Quote operation, as described in [18].

- A TNCS that supports IF-TNCCS-SOH 1.0 [12] allows an IMV to use IF-TNCCS-SOH features that would not otherwise be supported by IF-IMV: extra fields in SSoH and SSoHR and SOHReportEntries and SOHRReportEntries, etc. The most common operations will be simple (e.g. sending and receiving a vendor-specific message and providing a recommendation that is automatically translated into an SoHRReportEntry and SSoHR).

- A TNCS that supports IF-TNCCS 2.0 allows an IMV to use IF-TNCCS 2.0 features that were not supported by previous versions of IF-IMV: TNCS sending the first message in a handshake, multiple language preferences, and IMVs sending and receiving IMC-IMV messages with extended values (8 flag bits, 32 bit subtype, and source and destination IMC ID).

- An IMV can determine which IF-TNCCS and IF-T protocol and version are being used for a particular connection.

- An IMV can receive from the TNCS information about the authenticated identity and authentication method used by the Access Requestor for a particular connection. This information (which may include user identity, machine identity, or both) can be used for many purposes, such as correlating current integrity measurements with previous measurements from the same AR or using the identity to help determine which sets of policies should apply.

2.4 Unsupported Use Cases

Several use cases, including but not limited to this one, are not covered by (but not prevented by) this version of IF-IMV:

- None identified

2.5 Requirements

The following are the requirements which IF-IMV must meet in order to successfully play its role in the TNC architecture. These are stated as general requirements, with specific requirements called out as appropriate.

a. Meets the needs of the TNC architecture

IF-IMV must support all the functions and use cases described in the TNC architecture as they apply to the relationship between the TNC Server and IMV components.

Specific requirements include:

- The API must support multiple overlapping network connections and Integrity Check Handshakes for a single TNCS from multiple TNCCs, and communication between the TNCS and multiple IMVs.
- The API must allow an IMV to act as a front end for one or more back-end applications or remote servers, or not to act as a front end at all, as determined by the IMV implementer.
- IF-IMV must have some mechanism for IMVs to recommend isolation and compliance information to the TNCS, so that isolation can properly be supported on

the network. This may stop short of an explicit mechanism for knowing which network to assign for isolation, but there must be a way to pass intelligence from IMVs to the TNCS.

- IMVs MUST be able to recommend initiation of an Integrity Check Handshake retry.

b. Secure

The integrity and confidentiality of communications between an IMC and an IMV must be protected. The TNC Client and TNC Server are assumed to provide a secure communications tunnel between the IMCs and the IMVs. The IMCs and IMVs may choose to add other security mechanisms, but those are out of scope for this document.

Specific requirements include:

- The security considerations include requirements that unauthorized parties cannot observe communications between the IMV and the TNC Server; that only authorized IMVs can communicate with the TNC Server across IF-IMV and thence to the IMCs; and that no party can cause denial of service to any of the system components. See the Security Considerations section of this document for detailed discussion.

c. Efficient

The TNC architecture delays network access until the endpoint is determined to not pose a security threat to the network based on its asserted integrity information. To minimize user frustration, it is essential to minimize delays and make IMC-IMV communications as rapid and efficient as possible. Efficiency in IF-IMV is also important when considering that TNCSs and IMVs are server-side components which may be required to handle messages from thousands to millions of remote clients.

d. Extensible

IF-IMV needs to expand over time as new features are added to the TNC architecture.IF-IMV must allow new features to be added easily, providing for a smooth transition and allowing newer and older architectural components to continue to work together.

e. Scalable

IF-IMV is an interface in a critical server-side architecture and must support scalability levels appropriate to this role. Enterprise and service provider deployments of TNC server architectures may be required to support up to millions of clients and a corresponding load of message transactions. IF-IMV should allow TNC Servers to employ load balancing, failover, and other techniques to achieve scalability.

f. Reliable

Reliability of network operations is critical. IF-IMV must support reliable communications, as well as reliable implementations and deployments of TNCSs and IMVs. For example, it should be possible for vendors to implement redundancy features.

g. Easy to use and implement

IF-IMV should be easy for TNC Server and IMV vendors to use and implement. It should allow them to enhance existing products to support the TNC architecture and integrate legacy code without requiring substantial changes. IF-IMV should also make things easy for system

administrators and end-users. Components of the TNC architecture should plug together automatically without requiring extensive manual configuration.

h. Platform-independent

Since there is a wide variety of platforms which are deployed in server-side systems, IF-IMV must function on as many server platforms as possible. At least Java, Windows, Linux (most common flavors), and other UNIX variants must be supported. Platform bindings included in this specification describe how platform-specific issues are handled.

i. Language-independent

IF-IMV must support the widest possible variety of programming languages: C, C++, C#, Java, Visual Basic, assembly language, and others. Therefore, this specification defines an abstract API and language-specific bindings. All language-specific bindings are required to support all capabilities of the abstract API.

j. Allow Java IMVs and TNCSs to contain or interface with native code

TNC components that use the Java Platform Binding may need to include or interface with native (non-Java) code. The Java Platform Binding should not include any explicit support for this but it should not prevent it either.

k. Internationalized

IF-IMV must be able to support a wide variety of human languages. In particular, IF-IMV must enable IMVs to provide reason strings in the endpoint user's preferred language(s).

2.5.1 Non-Requirements

There are certain requirements that IF-IMV explicitly is not required to meet. This list may not be exhaustive (complete).

a. There is **no requirement** that IF-IMV provide *explicit* mechanisms for redundancy and failover. It is acceptable that vendor IMVs and TNCSs are able to provide proprietary redundancy and failover mechanisms.

b. There is **no requirement** that IF-IMV provide support for several TNCSs from different vendors running in a single Java Virtual Machine at the same time.

2.6 Assumptions

Here are the assumptions that IF-IMV makes about other components in the TNC architecture.

- Secure Message Transport

  The TNC Client and TNC Server are assumed to provide a secure communications tunnel for messages sent between the IMCs and the IMVs.

- Reliable Message Delivery

  The TNC Client and TNC Server are assumed to provide reliable delivery for messages sent between the IMCs and the IMVs. In the event that reliable delivery cannot be provided, the TNC Client or TNC Server is expected to terminate the connection.

- TNCS provides Action Recommendations for access decision

It is assumed that the TNCS combines IMV Action Recommendations from multiple IMVs (using whatever logic) and provides a final TNCS Action Recommendation to the entity which makes the access decision. Outside the scope of this specification, there is assumed to be a mechanism or mechanisms for the TNCS to thus communicate with this entity (which may include, for example, NAAs and other PDP components). However, this mechanism is not part of IF-IMV and will not be specified in this document. This may be defined in a future phase of the TNC specifications process. Implementers are encouraged to become familiar with the TNC architecture [1], which includes a detailed discussion of these entities and interactions.

- Statefulness/Statelessness

TNCSs and IMVs may be stateless with respect to any individual TNCCs/IMCs, or they may keep state. This is an implementation decision, not a requirement of the interface.

2.7 Keywords

The key words “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “MAY”, and “OPTIONAL” in this document are to be interpreted as described in RFC 2119 [2]. This specification does not distinguish blocks of informative comments and normative requirements. Therefore, for the sake of clarity, note that lower case instances of must, should, etc. do not indicate normative requirements.

2.8 Abstract API Naming Conventions

To avoid name conflicts, all identifiers in the IF-IMV Abstract API have a name that begins with “`TNC_`”. Note that this only pertains to the IF-IMV Abstract API. Since Java includes good support for scoped names, the Java Platform Binding often omits this prefix.

Functions described in this document that are to be implemented by an IMV have a name that begins with “`TNC_IMV_`”. This prefix is followed by words describing the operation performed by the function.

Functions described in this document that are to be implemented by a TNC Server (known as “callbacks") have a name that begins with “`TNC_TNCS_`”. This prefix is followed by words describing the operation performed by the function.

Vendor-specific functions MUST have a name that begins with “`TNC_XXX_`” where `XXX` is replaced by the vendor ID of the organization that defined the extension. See section 3.2.4 for more information and requirements on vendor-specific functions.

2.9 Features Provided by IF-IMV

This section documents the features provided by IF-IMV.

2.9.1 Integrity Check Handshake

One of the primary functions of IF-IMV is to enable message exchanges between IMCs and IMVs to share security state allowing the IMVs to factor the integrity of the IMC’s security software state into the access control decision. These communications always take place within the context of an *Integrity Check Handshake*. In such a handshake, the IMCs send a batch of messages (typically, integrity measurements) to the IMVs and the IMVs optionally respond with a batch of messages (remediation instructions, queries for more information, etc.). This dialog may go on for some time until the IMVs decide on their Action Recommendations.

2.9.2 Connection Management

A connection between a TNCC and a TNCS may include several Integrity Check Handshakes: an initial handshake that ends with the endpoint being told to perform remediation such as applying patches (which may involve rebooting the endpoint), a subsequent handshake once the remediation is complete, and sometimes even later handshakes such as when policies change. Handshakes for a given TNCC-TNCS pair cannot be nested. One such handshake must end before another can begin. To optimize and manage handshakes, the TNCS provides connection management features.

When a new TNCC-TNCS relationship is established, the TNCS chooses a network connection ID to refer to that relationship. The TNCS informs the IMVs of the new network connection and updates them whenever the state of the network connection changes. When a network connection is complete, the TNCS notifies the IMVs that the network connection ID will be deleted and then does so. Note that the connection ID is local to the TNCS (like a socket descriptor in UNIX), not shared with the TNCC.

A TNC MAY maintain the same network connection ID across several Integrity Check Handshakes between a particular TNCC-TNCS pair. There are two reasons to maintain a network connection ID beyond a single Integrity Check Handshake. First, this allows the IMCs and IMVs to maintain state information associated with an earlier handshake, avoiding the need to resend data if it was sent in an earlier handshake and has not changed. Second, it allows an IMV to request a handshake retry for a particular connection, as when policies change. The TNCS MAY ensure that connection IDs persist long enough to permit handshake retry but this is purely optional. In contrast, TNCCs SHOULD retain connection IDs so that handshakes can be automatically retried after remediation is complete. It may seem problematic to have a TNCC retain its connection ID for a connection and not have the TNCS retain its connection ID for that connection. This does not actually cause problems since the connection IDs are local identifiers (like a socket number) and are not shared by the TNCC and TNCS. The TNCS MUST use the same connection ID for all IMVs when referring to a particular connection.

2.9.3 Remediation and Handshake Retry

In several cases, it is useful to retry an Integrity Check Handshake. First, an endpoint may be isolated until remediation is complete. Once remediation is complete, an IMC can inform the TNCC of this fact and suggest that the TNCC retry the Integrity Check Handshake. Second, a TNCS can initiate a retry of an Integrity Check Handshake (if the TNCS or IMV policies change or as a periodic recheck). Third, an IMC or IMV can request a handshake retry in response to a condition detected by the IMC or IMV (suspicious activity, for instance). In any case, it's generally desirable (but not always possible) to reuse state established by the earlier handshake and to avoid disrupting network connectivity during the handshake retry. IF-TNCCS 2.0 and IF-T Binding to TLS 1.0 support this feature.

To support handshake retries, the TNCS MAY maintain a network connection ID after an Integrity Check Handshake has been completed. This network connection ID can then be used by the TNCS to inform IMVs that it is retrying the handshake or by an IMV to request a retry (due to policy change or another reason).

Handshake retry may not always be possible due to limitations in the TNCC, NAR, PEP, or other entities. In other cases, retry may require disrupting network connectivity. For these reasons, IF-IMV supports handshake retry and requires IMVs to handle handshake retries (which is usually trivial) but does not require TNCSs to honor IMV requests for handshake retry. In fact, IF-IMV requires an IMV to provide information about the reason for requesting handshake retry so that the TNCS can decide whether it wants to retry (which may disrupt network access).

Note that remediation instructions are delivered from IMVs to IMCs through standard IMV-IMC messages (see section 2.9.4, "Message Delivery"). There is no special support in IF-IMV for this feature. IMVs SHOULD send remediation instructions to IMCs before returning an IMV Action

Recommendation and IMV Evaluation Result to the TNCS so the instructions are delivered before the handshake is completed.

2.9.4 Message Delivery

One of the critical functions of the TNC architecture is conveying messages between IMCs and IMVs. Each message sent in this way consists of a message body, a message type, and a recipient type.

The message body is a sequence of octets (bytes). The TNCC and TNCS SHOULD NOT parse or interpret the message body. They only deliver it as described below. Interpretation of the message body is left to the ultimate recipients of the message, the IMCs or IMVs. A zero length message is perfectly valid and MUST be properly delivered by the TNCC and TNCS just as any other IMC-IMV message would be.

The message type is a four octet number that uniquely identifies the format and semantics of the message. The method used to ensure the uniqueness of message types while providing for vendor extensions is described below.

The recipient type is simply a flag indicating whether the message should be delivered to IMVs or IMCs. Messages sent by IMCs are delivered to IMVs and vice versa. All messages sent by an IMV through IF-IMV have a recipient type of IMC. All messages received by an IMV through IF-IMV have a recipient type of IMV. The recipient type does not show up in IF-IMC or IF-IMV, but it helps in explaining message routing.

The routing and delivery of messages is governed by message type and recipient type. Each IMC and IMV indicates through IF-IMC and IF-IMV which message types it wants to receive. The TNCC and TNCS are then responsible for ensuring that any message sent during an Integrity Check Handshake is delivered to all recipients that have a recipient type matching the message's recipient type and that have indicated the wish to receive messages whose type matches the message's message type. If no recipient has indicated a wish to receive a particular message type, the TNCC and TNCS can handle these messages as they like: ignore, log, etc.

WARNING: The message routing and delivery algorithm just described is not a one-to-one model. A single message may be received by several recipients (for example, two IMVs from a single vendor, two copies of an IMC, or nosy IMVs that monitor all messages). If several of these recipients respond, this may confuse the original sender. IMCs and IMVs MUST work properly in this environment. They MUST NOT assume that only one party will receive and/or respond to a message.

IF-IMV allows an IMV to send and receive messages using this messaging system. Note that this system should not be used to send large amounts of data. The messages will often be sent through PPP or similar protocols that do not include congestion control and are not well suited to bulk data transfer. If an IMC needs to download a patch (for instance), the IMV should indicate this by reference in the remediation instructions. The IMC will process those instructions after network access (perhaps isolated) has been established and can then download the patch via HTTP or another appropriate protocol.

All messages sent with `TNC_TNCS_SendMessage` and received with `TNC_IMV_ReceiveMessage` are between the IMC and IMV. The IMV communicates with the TNCS by calling functions (standard and vendor-specific) in the IF-IMV, not by sending messages. The TNCS should not interfere with communications between the IMC and IMVs by consuming or blocking IMC-IMV messages.

A particular example of the message delivery provided by IF-IMV is the communication of remediation instructions from the IMVs through the TNCS to the TNCC/IMCs. This is one application of IMC-IMV message delivery and in all cases follows the normal IMV-IMC communications path. IF-IMV provides support for communicating remediation instructions to an endpoint using this mechanism. Since the normal IMC-IMV communications path is used to

communicate remediation instructions, this specification will not address further the details of how remediation itself is done.

2.9.5 Reliability

For successful enterprise deployments, reliability of TNCSs and IMVs is important. To ensure this reliability, organizations may employ redundant TNCSs. Organizations may also require active failover as well as other features that provide a level of high availability for critical networks. Vendors and enterprises wishing to implement their systems incorporating redundancy should see the discussion of this topic in the TNC Architecture document [1].

2.9.6 Batches

IMC-IMV messages will frequently be carried over protocols (like EAP) that require participants to take turns in sending ("half duplex"). To operate well over such protocols, the TNCC sends a batch of messages and the TNCS responds with some messages.

To simplify the development of IMCs and IMVs, IF-IMC and IF-IMV always group IMC-IMV messages into batches. With previous versions of IF-IMC and IF-IMV, IMCs always send the first batch of messages. Starting with this version, either IMCs or IMVs can send the first batch of messages. The receiver can then respond with another batch of messages and the communication can thus continue. Regardless of whether the underlying protocol is half duplex, the TNCC and TNCS still must send IMC-IMV messages in batches and take turns in delivering those messages.

An IMV can only send a message in three circumstances: during the initial batch (when `TNC_IMV_BeginHandshake` is called), in response to a message received by the IMV in a batch (when `TNC_IMV_ReceiveMessage`, `TNC_IMV_ReceiveMessageSOH`, or `TNC_IMV_ReceiveMessageLong` is called), and at the end of a batch (when `TNC_IMV_BatchEnding` is called). In any of these circumstances, the IMV MAY send one or more messages by calling `TNC_TNCS_SendMessage`, `TNC_TNCS_SendMessageSOH`, or `TNC_TNCS_SendMessageLong` once for each message to be sent and then returning from `TNC_IMV_BeginHandshake`, `TNC_IMV_ReceiveMessage`, `TNC_IMV_ReceiveMessageSOH`, `TNC_IMV_ReceiveMessageLong`, or `TNC_IMV_BatchEnding`. Note that if the IMV does not call `TNC_TNCS_SendMessage`, `TNC_TNCS_SendMessageSOH`, or `TNC_TNCS_SendMessageLong` before returning from `TNC_IMV_BeginHandshake`, `TNC_IMV_ReceiveMessage`, `TNC_IMV_ReceiveMessageSOH`, `TNC_IMV_ReceiveMessageLong` or `TNC_IMV_BatchEnding`, this indicates that it does not want to send any messages at this time. IMCs use a similar mechanism.

If no IMCs want to send a message in a particular batch, the TNCC and TNCS will proceed to complete the handshake. Similarly, if no IMVs want to send a message in a particular batch, the TNCC and TNCS will proceed to complete the handshake. Therefore, an IMV that is not engaged in a dialog with an IMC may well find that the handshake has ended.

To deliver IMC messages to IMVs, the TNCS calls `TNC_IMV_ReceiveMessage`, `TNC_IMV_ReceiveMessageSOH`, or `TNC_IMV_ReceiveMessageLong`. The IMV may process the message immediately or queue it for later processing. However, if the IMV wants to send a message in response, it must do so by calling the `TNC_TNCS_SendMessage`, `TNC_TNCS_SendMessageSOH`, or `TNC_TNCS_SendMessageLong` function before returning from `TNC_IMV_ReceiveMessage`, `TNC_IMV_ReceiveMessageSOH`, or `TNC_IMV_ReceiveMessageLong`. Once all IMVs have finished sending their messages for a batch, the TNCS will send those messages to the TNCC and await its response. When this response is received, the TNCS will deliver to IMVs any messages sent by IMCs and start accepting messages from IMVs. These various receive and send messages may be mixed subject to constraints described in more detail below.

As with all IMV functions, the IMV SHOULD NOT wait a long time (definitely not more than a second) before returning from TNC_IMV_BeginHandshake, TNC_IMV_ReceiveMessage, TNC_IMV_ReceiveMessageSOH, TNC_IMV_ReceiveMessageLong, or TNC_IMV_BatchEnding. A long delay might frustrate users or exceed network timeouts (PDP, PEP or otherwise). IMVs that need to perform a lengthy process may want to simply send a status message, indicating that they are working. The IMCs can respond in the next batch with a status query and thus the handshake can be kept going.

Similarly, an IMV might expect to receive a “working” status message from an IMC during a particular batch, and if so can respond in the next batch with a status query to the IMC to keep that handshake going.

Note that a TNCC or TNCS MAY cut off IMC-IMV communications at any time for any reason, including limited support for long conversations in underlying protocols, user or administrator intervention, or policy. If this happens, the TNCS will return TNC_RESULT_ILLEGAL_OPERATION from TNC_TNCS_SendMessage, TNC_TNCS_SendMessageSOH, or TNC_TNCS_SendMessageLong. If the TNCS supports limiting the message size or number of round trips, the TNCS MUST return TNC_RESULT_EXCEEDED_MAX_MESSAGE_SIZE or TNC_RESULT_EXCEEDED_MAX_ROUND_TRIPS respectively if the limits are exceeded. An IMV can get the Maximum Message Size and Maximum Number of Round Trips by using the related Attribute ID within the TNC_TNCS_GetAttribute function. The IMV SHOULD adapt its behavior to accommodate these limitations if available.

2.9.7 IMV Action Recommendation

One of the assumptions of the TNC architectural model is that IF-IMV provides a means for IMVs to recommend action information to the TNCS, so that isolation can properly be supported on the network. The TNCS then will combine these IMV Action Recommendations using some logic (defined by the TNCS implementers) to come up with an overall TNCS Action Recommendation. Note that the TNCS may choose to ignore any IMV Action Recommendation, but each IMV must be able to recommend an action. Potential choices for IMV Action Recommendations include: recommend full (normal) access; recommend isolation (limited or quarantined access); and recommend denial (no access). The mandatory function TNC_TNCS_ProvideRecommendation is the mechanism within IF-IMV for an IMV to indicate its IMV Action Recommendation.

2.9.8 Reason String

A new feature of IF-IMV 1.2 is that an IMV can supply a reason string to explain its IMV Action Recommendation. In order to preserve backward compatibility with older TNCSs, this is an optional feature. An IMV MUST use dynamic function binding (where present) to determine whether a TNCS supports this feature.

The format of the reason string is not defined. It is simply a UTF-8 string. This provides maximum flexibility to the IMV in creating the reason string. However, it is suggested that the reason string be short but informative. When creating reason strings, remember that the user may not have network access. Some IMVs may allow the system administrator to configure the reason string. Others may provide the reason string themselves.

The TNCC and TNCS MAY choose to log the reason string, ignore it, display it to the endpoint user, combine it with other reason strings, or take any other action with it. They MAY modify the reason string such as removing or display control characters or display or truncating long strings.

The IMV SHOULD try to accommodate the language preferences conveyed via the TNC_TNCS_GetAttribute function with the AttributeID set to **TNC_ATTRIBUTEID_PREFERRED_LANGUAGE**. This can allow the endpoint user to view the reason string in their native language. Of course, it is understood that every IMV will have limits to the language preferences it can accommodate. Some IMVs will have only one language supported. This is acceptable although not optimal.

2.9.9 Stateless IMVs

A simple IMV (as described in section 8) can avoid maintaining per-IMC state. Such an IMV (known as a “stateless IMV”) might receive two IMC messages in a single handshake (as when two IMCs that send the same message are configured on one TNCC). This would cause the IMV to provide two IMV Action Recommendations for a single handshake, which might confuse the TNCS. A TNCS SHOULD be prepared to receive more than one IMV Action Recommendation from an IMV for a single handshake. The TNCS MAY handle these multiple IMV Action Recommendations in any way: ignoring the first, ignoring the last, combining them, logging a message, refusing access, or anything else.

2.9.10 IMVs with Remote Servers

As an implementation choice, an IMV may consist of a “stub” DLL located on the TNCS host. This stub can talk a vendor-specific protocol to back-end remote servers which implement, for example, integrity verification or policy management functions. A “stub” IMV presents the full IF-IMV interface, and may convert from IF-IMV interface to the vendor specific protocol. In this case, it is of course the responsibility of the IMV vendors to provide the “stub” as well as any remote server. Any redundancy or failover – indeed, all IMV functionality whatsoever – must be provided within the “stub” and its vendor-specific protocol. IMVs also may, of course, be “standalone” and collocated with the TNCS. In either case, the IMV is defined as that entity which speaks IF-IMV to the TNCS, regardless of whether any remote server also exists.

3 IF-IMV Abstract API

The IF-IMV Abstract API defines a small number of standard functions that an IMV can implement. The TNC Server calls these functions when it needs the IMV to perform an action (such as processing a message from an IMC). The API also defines certain functions that the TNC Server implements (known as “callbacks”). The IMV calls these functions when it needs the TNC Server to perform an action (such as sending a message to an IMC).

3.1 Platform and Language Independence

IF-IMV is a language-independent abstract API. It can be mapped to almost any programming language. This section defines the abstract API, using C syntax (as defined in [8]) for ease of comprehension. Because different languages have different conventions and constructs (functions, objects, etc.), the abstract API may need to be modified for different languages in different bindings. However, this should be avoided as much as possible to increase compatibility between IMVs and TNCSs written in different languages.

Section 6 provides a C header file that serves as a binding for the C language with the Microsoft Windows DLL platform binding. The Java Platform Binding in section 4.3 provides a binding for the Java Programming Language. Bindings for other programming languages may be defined in the future. However, many languages can use or implement libraries with C bindings. Implementers SHOULD use the C language binding when possible for maximum compatibility with other IMVs and TNC Servers on their platform. This specification does not provide a standard way to mix an IMV written in one language with a TNCS written in another language, beyond the support that may be provided by platform-specific bindings.

IF-IMV is also a platform-independent API. It is designed to support almost any platform. Platform-specific bindings are described in section 3.9.8. The IF-IMV API definition sometimes uses language like “unsigned integer of at least 32 bits.” To see the exact definition of this for a particular platform (operating environment and/or language), see the platform-specific bindings.

3.2 Extensibility

To meet the Extensibility requirement defined above, the IF-IMV API includes several extensibility mechanisms: an API version number, dynamic function binding, and vendor IDs.

3.2.1 API Version

This document defines version 1 of the TNC IF-IMV API. Future versions may be incompatible due to removing, adding, or changing functions, types, and constants. However, the `TNC_IMV_Initialize` function and its associated types and constants will not change so that version incompatibilities can be detected. A TNCS or IMV can even support multiple versions of the IF-IMV API for maximum compatibility. See section 3.8.1 for details.

3.2.2 Dynamic Function Binding

Platforms that support IF-IMV SHOULD support dynamic function binding. This feature allows a TNCS or IMV to define functions that go beyond those included in this API and allows the other party to determine whether those functions are defined, call them if so, and handle their absence gracefully. Dynamic function binding is needed to support optional and vendor-specific functions and so that a TNCS or IMV can support multiple API versions.

On platforms that don’t define a Dynamic Function Binding mechanism, all optional functions MUST be implemented, vendor-specific functions MUST NOT be implemented or used except by private convention, and provisions must be made to insure that TNCSs and IMVs that support different version numbers interact safely.

3.2.3 Vendor IDs

The IF-IMV API supports several forms of vendor extensions. IMV or TNCS vendors can define vendor-specific functions and make them available to the other party. IMV or TNCS vendors can define vendor-specific result codes and vendor-specific attribute IDs. And IMV vendors can define vendor-specific message types (for the messages sent between IMCs and IMVs).

In each of these cases, SMI Private Enterprise Numbers are used to provide a separate identifier space for each vendor. IANA provides a registry for SMI Private Enterprise Numbers at http://www.iana.org/assignments/enterprise-numbers. Any organization (including non-profit organizations, governmental bodies, etc.) can obtain one of these numbers at no charge and thousands of organizations have done so. Within this document, SMI Private Enterprise Numbers are known as “vendor IDs”.

In previous versions of this specification, vendor ID 0 was reserved for the TCG. However, it has been determined that this value is actually reserved for the IETF. Therefore, TCG will use its actual SMI Private Enterprise Number, 0x005597, for new allocations. Existing allocations with vendor ID 0 will remain in place. This should not be a problem, since none of the existing allocations with vendor ID 0 pertain to network protocols, only to numbers internal to the IF-IMV interface.

Some people have raised concerns that 24 bits is not enough for a vendor ID. This is not a significant concern since the IANA allocates vendor IDs in sequential order, and they have only allocated about 40,000 vendor IDs in the last 20 years. At that rate, this won’t become a problem for another 8,000 years!

Vendor ID 16777215 (0xffffff) is reserved for use as a wildcard. For details of how vendor IDs are used to support vendor-specific functions, result codes, and message types, see sections 3.2.4, 3.5.2.14, and 3.5.2.7.

3.2.4 Vendor-Specific Functions

The IMV and TNC Server MAY extend the IF-IMV API by defining vendor-specific functions that go beyond those described here. An IMV or TNC Server MUST work properly if a vendor-specific function is not implemented by the other party and MUST ignore vendor-specific functions that it does not understand. To determine whether a vendor-specific function has been implemented, use the dynamic function binding mechanism defined in the platform binding.

Vendor-specific functions MUST have a name that begins with “`TNC_XXX_`” where `XXX` is replaced by the vendor ID of the organization that defined the extension. The vendor ID is converted to ASCII numbers or the equivalent, using a decimal representation whose initial digit MUST NOT be zero (0). For instance, the organization owning the vendor ID 1 could define a vendor-specific function named “`TNC_1_ProcessMapping`”. Avoid defining names longer than 31 characters since some platforms do not support such long names well. If a vendor-specific function is designed to be implemented by only one TNC component, then it is helpful to put the name of this component in the function name after the vendor ID. For instance, a function named “TNC_1_IMV_Reinstall” is clearly intended to be implemented by IMVs.

3.3 Protocol Independence and Adaptation

The TNC architecture tries to insulate IMCs and IMVs so that they do not need to be aware of different underlying protocols (IF-TNCCS and IF-T). However, some IMCs and IMVs want to change their behavior depending on the underlying protocols and sometimes limitations of the underlying protocols (such as limited data capacity or round trips) intrude on the operation of IMCs and IMVs. For these reasons, IF-IMV now includes several functions and features that allow IMVs to adapt their behavior to underlying protocols. This section enumerates those features. It also describes how TNCSs are expected to behave with each of the underlying protocols, since their behavior must be predictable to ensure interoperability.

3.3.1 IMV Adaptation to Lower Layer Protocols

Simple IMVs may not need to adapt their behavior to differences in lower layer protocols. For example, an IMV that just accepts an integrity message and recommends an action should not need to adapt. However, an IMV that wants to engage in a multi-round handshake by requesting additional integrity information or an IMV that sends a lot of data may need to adapt since some protocols only support one round trip and a small amount of data. When such a protocol is in use, an IMC may adapt by just sending a short summary report allowing the corresponding IMV to recommend action with no need for further communication.

IF-IMV provides several capabilities that an IMV can use to adapt its behavior to underlying protocols. An IMV implementer may choose to employ any or all of these capabilities. However, the IMV implementer must remember that many TNCSs do not support these capabilities. IMVs MUST work properly if they're not supported.

- An IMV can read attributes that provide generic information about a connection. The Maximum Round Trips attribute indicates the maximum number of round trips supported by the underlying protocols. The Maximum Message Size attribute indicates the maximum message size supported by the underlying protocols. Other attributes (Has Long Types, Has Exclusive, and Has SOH) indicate whether the underlying protocols support certain specific features. Generic attributes are especially nice because they allow an IMV to adapt its behavior in a generic manner to whatever the underlying protocols may be.
- An IMV can use protocol-specific functions or attributes to take advantage of features that are specific to a particular protocol. For example, the `TNC_TNCS_SendMessageSOH` function allows an IMV to send fields that are specific to the IF-TNCCS-SOH protocol.
- An IMV can read protocol type and version attributes and adapt its behavior to one specific protocol version. For example, an IMV that detects the IF-TNCCS 2.0 protocol may accept messages in the IF-M 1.0 format.

The IF-IMV interface still has a goal of insulating IMVs from differences in underlying protocols. For simple IMVs, this works well. But more sophisticated IMVs can now take full advantage of the features of underlying protocols in a manner that ensures compatibility with a wide variety of TNCSs.

3.3.2 TNCS Behavior With Different Lower Layer Protocols

TNCSs need to change their behavior when different versions of IF-TNCCS are used. For example, a TNCS needs to use different message formats for different versions of IF-TNCCS. Most of these behavior changes are described in the specifications for the various IF-TNCCS versions. This section provides an overview of behavior changes that pertain to IF-IMV. Further requirements are contained in the rest of this specification (e.g. in the description of functions like `TNC_IMV_ReceiveMessageSOH`).

3.3.2.1 TNCS Behavior with IF-TNCCS-SOH 1.0

The IF-TNCCS-SOH 1.0 protocol presents several challenges. First, it only supports one round trip and the total message size for the SoH or SoHR message is limited to 4000 bytes. Second, there are many fields in the SoH and SoHR messages that are not noted in the TNC architecture. This section provides specific requirements and recommendations for how these challenges can be addressed.

The challenge of limited round trips and message size is addressed by exposing the limitations of the protocol to the IMVs so that they can adapt and enforce those limitations if the IMVs do not adapt. A TNCS that implements IF-TNCCS-SOH and allows IMVs to participate in handshakes using this protocol SHOULD implement the Maximum Round Trips attribute, setting its value to 1 for connections that use IF-TNCCS-SOH, and SHOULD implement the Maximum Message Size attribute, setting its value to a number chosen by the TNCS to ensure that if each IMVs complies with this constraint then the total size of the SoHR message will not exceed 4000 bytes. This document does not specify the exact method used to calculate the value of the Maximum

Message Size attribute. One simple method is to subtract the expected message overhead (including SoHR header, SSoHRs, etc.) from 4000 and divide by the number of IMVs. A more sophisticated method might provide lower values to IMVs that generally only send short messages, thereby allowing other IMVs to send more data.

The challenge of the many fields in the SoH and SoHR messages is addressed by providing SOH-specific functions and attributes that IMVs can use to access these fields: `TNC_IMV_ReceiveMessageSOH`, `TNC_TNCS_SendMessageSOH`, and the Has SOH and SSoH, and SOH attributes. A TNCS that allows IMVs to participate in handshakes using the IF-TNCCS-SOH protocol SHOULD implement these functions and attributes. In a future version of this specification, this recommendation may be upgraded to a requirement (MUST). The next few paragraphs describe how a TNCS should implement support for IF-TNCCS-SOH, including the use of these functions and attributes. These paragraphs only apply when IF-TNCCS-SOH is used for a connection.

When a TNCS is preparing to start an IF-TNCCS-SOH handshake, its first step should be to set the Has SOH attribute for the connection. After this, the TNCS's behavior is just the usual: call the TNC_IMV_NotifyConnectionChange function for IMVs that are participating in the handshake with the value TNC_CONNECTION_STATE_HANDSHAKE and deliver messages sent by IMCs. There are a few differences with IF-TNCCS-SOH though. One difference is that IMC messages SHOULD be delivered using `TNC_IMV_ReceiveMessageSOH` if the IMV implements it otherwise `TNC_IMV_ReceiveMessage` should be used. Another difference is that the TNCS SHOULD allow the IMV to get the Maximum Round Trips, Maximum Message Size, and Has SOH attributes. Also, the TNCS SHOULD implement the `TNC_TNCS_SendMessageSOH` function so that the IMV can send full SOHRReportEntry messages. Finally, the TNCS MUST modify the behavior of the `TNC_TNCS_SendMessage` function so that each message sent by an IMV using this function is stuffed into a Vendor-Specific atrribute in an SOHRReportEntry message, as described in the description of the `TNC_TNCS_SendMessage` function. When all the IMVs' messages have been sent, the TNCS should send the SOHR message.

Note that the TNCS is responsible for creating the SSoHR and inserting it into the SOHR message. The method that the TNCS uses to create the SSoHR is not defined in this specification. It does not need to be specified since the SSoHR is not visible to the IMVs. IMCs can read the SSoHR via IF-IMC but that does not place any requirements on how the TNCS creates the SSoHR.

When a TNCS receives a SoH message, it should prepare the message so that IMVs can access it with the SOH and SSOH attributes and then parse the SOH message into SOHReportEntry messages. For each of these messages, the value contained in the first System-Health-ID attribute should be compared to the `TNC_MessageType` values previously supplied in the IMVs' most recent calls to `TNC_TNCS_ReportMessageTypes` or `TNC_TNCS_ReportMessageTypesLong`. The SOHReportEntry should be delivered to each IMV that has a match. If an IMV does not support `TNC_IMV_ReceiveMessageSOH` (or when the TNCS does not support that function), the TNCS should extract the Data field of the first Vendor-Specific attribute whose Vendor ID matches the value contained in the System-Health-ID and deliver that as the message using `TNC_IMV_ReceiveMessage`. If an IMV does support `TNC_IMV_ReceiveMessageSOH`, the TNCS should deliver the entire SOHReportEntry using that function. The TNCS should then call `TNC_IMV_BatchEnding` to indicate that the batch is ending and `TNC_IMV_NotifyConnectionChange` to indicate what the result of the handshake was.

3.3.2.2 TNCS Behavior with IF-TNCCS 2.0

The IF-TNCCS 2.0 protocol presents a different set of challenges. IF-TNCCS 2.0 supports long message types, exclusive delivery, TNCS preferred language, and the TNCS sending the first message in a handshake. This section describes how a TNCS that supports the IF-TNCCS 2.0 protocol should implement the IF-IMV interface so that IMVs know what to expect.

A TNCS that implements the IF-TNCCS 2.0 protocol MUST also implement the Has Long Types and Has Exclusive attributes, returning a value of 1 for each of these if the underlying protocol is IF-TNCCS 2.0. Such a TNCS (i.e. a TNCS that implements the IF-TNCCS 2.0 protocol) MUST also implement the `TNC_TNCS_SendMessageLong` and `TNC_TNCS_ReserveAdditionalIMVID` functions and call the `TNC_IMV_ReceiveMessageLong` function as described in the description of that function. And such a TNCS MUST implement the attributes Preferred Language, IF-TNCCS Protocol Name, IF-TNCCS Protocol Version, IF-T Protocol Name, and IF-T Protocol Version, returning the appropriate values. A TNCS that implements the IF-TNCCS 2.0 protocol MUST determine which IMVs implement the `TNC_IMV_BeginHandshake` API using dynamic function binding (on platforms where that is available) and call this function on such IMVs when the TNCS is soliciting the first batch of messages in the handshake.

When the IF-TNCCS 2.0 protocol is used, the TNCS MUST assign a 16-bit IMV Identifier to each IMV and send this in the IMV Identifier field of the TNCCS-IF-M-Message unless the IMV has reserved a different IMV Identifier with `TNC_TNCS_ReserveAdditionalIMVID` and passed this identifier to the `TNC_TNCS_SendMessageLong` function as the `sourceIMVID`. To simplify things, the TNCS MUST use the same value for this IMV identifier that it uses for the 32-bit imvID passed to all IMV functions. If a TNCS receives an IF-TNCCS 2.0 batch that contains a TNCCS-IF-M-Message with the `TNC_MESSAGE_FLAGS_EXCLUSIVE` flag ("the EXCL flag", for short) set, the TNCS SHOULD NOT deliver this message to any IMV other than the one whose IMV Identifier matches the IMV Identifier field of the TNCCS-IF-M-Message. If there is no such IMV or if that IMV has not reported an interest in the message type of the message, the TNCS should discard the message.

When the IF-TNCCS 2.0 protocol is used and the IMV implements `TNC_IMV_ReceiveMessageLong`, the TNCS SHOULD use this function to deliver messages instead of using `TNC_IMV_ReceiveMessage`. However, if the IMV does not implement `TNC_IMV_ReceiveMessageLong`, the TNCS SHOULD use `TNC_IMV_ReceiveMessage` instead. Messages whose vendor ID or message subtype is too long to be represented in the parameters supported by `TNC_IMV_ReceiveMessage` MUST NOT be delivered to this IMV.

3.4 Threading and Reentrancy

The TNCS MUST be reentrant (able to receive and process a function call even when one is already underway). IMV DLLs also MUST be reentrant.

The TNC Server and all IMV DLLs MUST be thread-safe. This means that any IF-IMV function can be called at any time even if other threads are also calling an IF-IMV function. The TNCS and IMVs may employ semaphores or other synchronization mechanisms to protect critical sections of code, but these mechanisms SHOULD be employed sparingly using best practices appropriate to the platform to maintain good performance in a highly multi-threaded server environment.

3.5 Data Types

3.5.1 Basic Types

These types are the most basic ones used by the IF-IMV API. They are defined in a platform-dependent and language-dependent manner to meet the requirements described in this section. Consult section 3.9.8 to see how these types are defined for a particular platform and language.

Type	Definition
`TNC_UInt32`	`Unsigned integer of at least 32 bits`
`TNC_BufferReference`	**`Reference to buffer of octets`**

3.5.2 Derived Types

These types are defined in terms of the more basic ones defined in section 3.5.1. They are described in the following subsections.

Type	**Definition**	**Usage**
TNC_IMVID	TNC_UInt32	IMV ID
TNC_ConnectionID	TNC_UInt32	Network Connection ID
TNC_ConnectionState	TNC_UInt32	Network Connection State
TNC_RetryReason	TNC_UInt32	Handshake retry reason
TNC_IMV_Action_Recommendation	TNC_UInt32	IMV Action Recommendation
TNC_IMV_Evaluation_Result	TNC_UInt32	IMV Evaluation Result
TNC_MessageType	TNC_UInt32	Message type
TNC_MessageTypeList	Platform-specific	Reference to list of TNC_MessageType
TNC_VendorID	TNC_UInt32	Vendor ID
TNC_VendorIDList	Platform-specific	Reference to list of TNC_VendorID
TNC_MessageSubtype	TNC_UInt32	Message subtype
TNC_MessageSubtypeList	Platform-specific	Reference to list of TNC_MessageSubtype
TNC_Version	TNC_UInt32	IF-IMV API version number
TNC_Result	TNC_UInt32	Result code
TNC_AttributeID	**TNC_UInt32**	**Attribute ID**

3.5.2.1 IMV ID

When a TNC Server loads an IMV, it assigns it an IMV ID (represented by the TNC_IMVID type). This allows the IMV to identify itself when calling TNCS functions. The IMV ID is a TNC_UInt32 chosen by the TNCS and passed to the TNC_IMV_Initialize function. This IMV ID is referred to as primary IMV ID in the document henceforth. It is valid until the TNCS calls TNC_IMV_Terminate for this IMV.

An IMV can also request additional IMV IDs, if the TNCS implements the TNC_TNCS_ReserveAdditionalIMVID function. The additional IMV IDs are valid until the TNCS calls TNC_IMV_Terminate for this IMV. This is useful when IF-M messages are used, as described in section 3.2 of IF-M 1.0 [20]. Refer to section 3.9.10 of this document (IF-IMV) for more details. Most of the IF-IMV functions implemented by either IMV or TNCS accept IMV ID as a parameter. This parameter MUST be the primary IMV ID of the IMV unless specified otherwise in the function documentation.

There is no internal structure to an IMV ID. The IMV ID TNC_IMVID_ANY(65535 or 0xffff) is a reserved value if the connection supports IF-TNCCS 2.0 as described in section 3.3.2.2. The TNCS can choose any other value for the IMV ID and the IMV MUST NOT attach any significance to the value chosen. However, the TNCS SHOULD NOT use values greater than 65535, since these cannot easily be accommodated in IF-TNCCS 2.0.

3.5.2.2 Network Connection ID

A TNCS will commonly be negotiating with several different TNCCs at once (when several endpoints are simultaneously conducting Integrity Check Handshakes). Each of these TNCC-TNCS pairs is referred to as a “network connection”.

To help the IMV track which IMC-IMV messages go with which network connection and perform other connection management tasks, the TNCS chooses a network connection ID (represented by the `TNC_ConnectionID` type) that identifies a particular network connection. This connection ID is local to the TNCS and not shared with the TNCC. It’s like a socket descriptor in UNIX. When a network connection is created, the TNCS chooses a network connection ID and then passes the network connection ID to the IMV as a parameter to the `TNC_IMV_NotifyConnectionChange` function with a `newState` of `TNC_CONNECTION_STATE_CREATE`. This informs the IMV that a new network connection has begun. The network connection ID then becomes valid.

The IMV and TNCS use this network connection ID to refer to the network connection when delivering messages and performing other operations relevant to the network connection. This helps ensure that IMV messages are sent to the right TNCC and IMCs, helps ensure that the IMV Action Recommendation is associated with the right endpoint, and helps the IMV match up messages from IMCs with any state the IMV may be maintaining from earlier parts of that IMC-IMV conversation (even extending across multiple Integrity Check Handshakes in a single network connection).

The TNCS notifies IMVs of changes in network connection state (handshake success, handshake failure, etc.) by calling the `TNC_IMV_NotifyConnectionChange` function. When a network connection is finished, the TNCS first notifies IMVs of this by calling the `TNC_IMV_NotifyConnectionChange` function with the network connection ID and a `newState` of `TNC_CONNECTION_STATE_DELETE`. The network connection ID then becomes invalid and any information associated with it can be deleted. Once a network connection enters the `TNC_CONNECTION_STATE_DELETE` state, it cannot transition to any other state.

As described in section 2.9.3 above, it is sometimes desirable to retry an Integrity Check Handshake (when remediation is complete, for instance). Some TNCSs will not support this but all IMVs MUST do so. To indicate that a network connection retry is beginning, a TNCS notifies the IMVs by calling the `TNC_IMV_NotifyConnectionChange` function with the network connection ID and a `newState` of `TNC_CONNECTION_STATE_HANDSHAKE`. This means that an Integrity Check Handshake will soon begin.

An IMV can ask the TNCS to retry an Integrity Check Handshake by calling the `TNC_TNCS_RequestConnectionRetry` function. For details on this, see the description of that function.

There is no internal structure to a network connection ID. There is one reserved value: `TNC_CONNECTIONID_ANY` (`0xffffffff`). The TNCS can choose any other value for a network connection ID that does not conflict with another valid network connection ID for the same TNCS-IMV pair. It can even choose a network connection ID that was used by a previous network connection that has now been deleted and is invalid. The IMV MUST NOT attach any significance to the value chosen. The network connection ID chosen by a TNCS for a particular network connection need not match the network connection ID chosen by the TNCC for that same connection. This is a local identifier only used between the TNCS and the IMVs.

3.5.2.3 Network Connection State

The TNCS uses the `TNC_IMV_NotifyConnectionChange` function to notify IMVs of changes in network connection state. The network connection state is represented as a `TNC_UInt32`. The TNCS MUST pass one of the values listed in section 3.6.5. The TNCS MUST NOT use any other network connection state value with this version of the IF-IMV API.

3.5.2.4 Handshake Retry Reason

The IMV can ask the TNCS to retry an Integrity Check Handshake by calling the `TNC_TNCS_RequestHandshakeRetry` function. One of the parameters to that function is a `TNC_RetryReason`. This type is represented as a `TNC_UInt32`. The IMV MUST pass one of the values listed in section 3.6.6. The IMV MUST NOT use any other handshake retry reason value with this version of the IF-IMV API.

3.5.2.5 IMV Action Recommendation

After evaluating the endpoint's integrity, each IMV supplies an IMV Action Recommendation and IMV Evaluation Result to the TNCS by calling the `TNC_TNCS_ProvideRecommendation` function. One call to TNC_TNCS_ProvideRecommendation suffices to pass both of these values. The type used to communicate the IMV Action Recommendation is `TNC_IMV_Action_Recommendation`. This type is represented as a `TNC_UInt32`. The IMV MUST pass one of the values listed in section 3.6.7. The IMV MUST NOT use any other IMV Action Recommendation value with this version of the IF-IMV API.

3.5.2.6 IMV Evaluation Result

After evaluating the endpoint's integrity, each IMV supplies an IMV Action Recommendation and IMV Evaluation Result to the TNCS by calling the `TNC_TNCS_ProvideRecommendation` function. One call to TNC_TNCS_ProvideRecommendation suffices to pass both of these values. The type used to communicate the IMV Evaluation Result is `TNC_IMV_Evaluation_Result`. This type is represented as a `TNC_UInt32`. The IMV MUST pass one of the values listed in section 3.6.8. The IMV MUST NOT use any other IMV Evaluation Result value with this version of the IF-IMV API. This document does not specify what the TNCS does with this value. It may log it.

3.5.2.7 Message Type

As described in section 2.9.4, the TNC architecture routes messages between IMCs and IMVs based on their message type. Each message has a message type that uniquely identifies the format and semantics of the message.

To ensure the uniqueness of message types while providing for vendor extensions, vendor-specific message types are formed out of a vendor-chosen message subtype and the vendor's vendor ID.

In previous versions of this specification, the message type was always placed into a single 32-bit number by placing the vendor ID in the most significant 24 bits of that number and the message subtype in the least significant 8 bits. However, there have been some complaints that this only allows each vendor to define 256 message subtypes. Therefore, a new expanded format for message types has been defined where the vendor ID and message subtype are each 32 bits in length. This new message type format is supported by new, optional functions in IF-IMV (`TNC_IMV_ReceiveMessageLong` and `TNC_TNCS_SendMessageLong`).

Of course, the new expanded format for message types will only work with software and protocols that support it. IMVs that wish to use the expanded format should use dynamic function binding to determine whether the TNCS implements the `TNC_TNCS_SendMessageLong` function. If so, they can try to use this function to send messages. If the connection does not support long message types and the message types passed to the function cannot be represented in 24 bits for the vendor ID and 8 bits for the message subtype, the result `TNC_RESULT_NO_LONG_MESSAGE_TYPES` will be returned.

The vendor ID `TNC_VENDORID_ANY` (`0xffffff`) and the subtype `TNC_SUBTYPE_ANY` (`0xff`) are reserved as wild cards as described in section 3.9.1. An IMV MUST NOT send messages whose message type includes one of these reserved values. When long message types are employed, these values are not the largest possible ones. Still, the same values are used as wild cards.

TNC Clients and TNC Servers MUST properly deliver messages with any message type that does not include a wild card (as described in section 2.9.4 and 3.9.1).

3.5.2.8 Message Type List

The TNC_MessageTypeList type represents a list of message types. The exact representation of this type is platform-specific, but it will typically be a pointer or reference to an array of TNC_MessageTypes.

3.5.2.9 Vendor ID

The TNC_VendorID type represents a vendor ID as described in section 3.2.3. This type may be used when forming and parsing message types. For a full description of vendor IDs, see section 3.2.3.

The message type TNC_VENDORID_ANY (0xffffff) is reserved as a wild card as described in section 3.9.1. IMVs may request messages with this vendor ID to indicate that they want to receive messages whose message type includes any vendor ID. However, an IMV MUST NOT send messages whose message type includes this reserved value and a TNCS MUST NOT deliver such messages. When long message types are employed, this value is not the largest possible one. Still, the same value is used as a wild card.

3.5.2.10 Vendor ID List

The TNC_VendorIDList type represents a list of vendor IDs. Its exact representation is platform-specific, but will typically be a pointer or reference to an array of TNC_VendorIDs.

3.5.2.11 Message Subtype

The TNC_MessageSubtype type represents a message subtype. This type may be used when forming and parsing message types. Note that message subtypes greater than 0xff only work with the new expanded message type format.

The message subtype TNC_SUBTYPE_ANY (0xff) is reserved as a wild card as described in section 3.9.1. IMVs may request messages with this message subtype to indicate that they want to receive messages whose message subtype has any value. However, an IMV MUST NOT send messages whose message subtype includes this reserved value and a TNCS MUST NOT deliver such messages. When long message types are employed, this value is not the largest possible one. Still, the same value is used as a wild card.

3.5.2.12 Message Subtype List

The TNC_MessageSubtypeList type represents a list of message subtypes. Its exact representation is platform-specific, but will typically be a pointer or reference to an array of TNC_MessageSubtypes.

3.5.2.13 Version

The TNC_Version type represents an API version number. See sections 3.2.1 and 3.8.1 for details on how this is used.

3.5.2.14 Result Code

Each function in the IF-IMV API returns a result code of type TNC_Result to indicate success or the reason for failure. As noted above, a result code is represented as a TNC_UInt32, an unsigned integer of at least 32 bits in length. To form a vendor-specific result code, place a vendor-chosen subcode in the least significant 8 bits of the integer and the vendor's vendor ID in the next most significant 24 bits of the result code (the most significant 24 bits if the integer is 32 bits long). Older result codes defined in this specification (listed in section 3.6.2) have the reserved value zero (0) in the most significant 24 bits. Newer ones have the value 0x005597 in the most significant 24 bits, since this is the TCG's official vendor ID.

IMVs and TNCSs MUST be prepared for any function to return any result code. Vendor-specific result codes are always permissible and new standard result codes may be defined without changing the version number of the IF-IMV API. Any unknown non-zero result code SHOULD be treated as equivalent to TNC_RESULT_OTHER.

3.5.2.15 Attribute ID

The TNC_AttributeID type identifies a TNC attribute. TNC attributes allow IMVs to set and get attribute values identified by a TNC_AttributeID and associated with a TNCS or network connection. For instance, an IMV can get the attribute value with attribute ID TNC_ATTRIBUTE_PREFERRED_LANGUAGE associated with a particular connection, which identifies the preferred language associated with that connection.

As noted above, an attribute ID is represented as a TNC_UInt32, an unsigned integer of at least 32 bits in length. As with result codes and message types, vendor-specific attribute IDs may be defined by particular vendors by placing a vendor-chosen subcode in the least significant 8 bits of the integer and the vendor's vendor ID in the next most significant 24 bits of the attribute ID (the most significant 24 bits if the integer is 32 bits long). The vendor who defines a particular vendor-specific attribute ID should carefully document the format of the attribute value for that attribute ID so that IMVs can properly use the attribute ID. Attribute IDs with a vendor ID of zero (0) or 0x005597 are reserved for definition by TCG. Some of these reserved attribute IDs are defined in section 3.6.11 below.

Generally, each TNCS will support only a limited set of attribute IDs. TNCSs MAY support no attribute IDs at all. IMVs MUST be prepared for this.

Note that the Java Platform Binding for IF-IMV uses objects instead of byte arrays for attribute values. Section 4.3.9 documents the objects used to represent attribute values for the reserved attribute IDs. Vendors who define vendor-specific attribute IDs SHOULD define what object is used to represent attribute values for those attribute IDs.

3.6 Defined Constants

This section describes the constants defined in the abstract IF-IMV API.

3.6.1 Boolean Values

There are only two permissible values of the type TNC_Boolean: TNC_TRUE and TNC_FALSE. These values are used to indicate Boolean values.

3.6.2 Result Code Values

Each function in the IF-IMV API returns a result code of type TNC_Result to indicate success or reason for failure. Here is the set of standard result codes defined by this specification. Vendor-specific result codes are always permissible and new standard result codes may be defined without changing the version number of the IF-IMV API. IMVs and TNCSs MUST be prepared for any function to return any result code. Any unknown non-zero result code SHOULD be treated as equivalent to TNC_RESULT_OTHER. IMCs or TNCCs MAY communicate errors to users, log them, ignore them, or handle them in another way that is compliant with this specification.

If an IMV function returns TNC_RESULT_FATAL, then the IMV has encountered a permanent error. The TNCS SHOULD call TNC_IMV_Terminate as soon as possible. The TNCS MAY then try to reinitialize the IMV with TNC_IMV_Initialize or try other measures such as unloading and reloading the IMV and then reinitializing it.

If a TNCS function returns TNC_RESULT_FATAL, then the TNCS has encountered a permanent error.

Result Code	Definition
TNC_RESULT_SUCCESS	Function completed successfully
TNC_RESULT_NOT_INITIALIZED	TNC_IMV_Initialize has not been called
TNC_RESULT_ALREADY_INITIALIZED	TNC_IMV_Initialize was called twice without a call to

	TNC_IMV_Terminate
TNC_RESULT_NO_COMMON_VERSION	No common IF-IMV API version between IMV and TNC Server
TNC_RESULT_CANT_RETRY	TNCS cannot attempt handshake retry
TNC_RESULT_WONT_RETRY	TNCS refuses to attempt handshake retry
TNC_RESULT_INVALID_PARAMETER	Function parameter is not valid
TNC_RESULT_ILLEGAL_OPERATION	Illegal operation attempted
TNC_RESULT_EXCEEDED_MAX_ROUND_TRIPS	Maximum round trips exceeded
TNC_RESULT_EXCEEDED_MAX_MESSAGE_SIZE	Maximum message size exceeded
TNC_RESULT_NO_LONG_MESSAGE_TYPES	Connection does not support long message types
TNC_RESULT_NO_SOH_SUPPORT	Connection does not support SOH
TNC_RESULT_OTHER	Unspecified error
TNC_RESULT_FATAL	**Unspecified fatal error**

3.6.3 Version Numbers

As noted in section 3.2.1, this specification defines version 1 of the TNC IF-IMV API. Future versions of this specification will define other version numbers. See section 3.8.1 for a description of how version numbers are handled.

Version Number	Definition
TNC_IMV_VERSION_1	**The version of IF-IMV API defined here**

3.6.4 Network Connection ID Values

The reserved value TNC_CONNECTIONID_ANY MUST NOT be used as a normal network connection ID. Instead, it may be passed to TNC_TNCS_RequestHandshakeRetry to indicate that handshake retry is requested for all current network connections.

Network Connection ID Value	Definition
TNC_CONNECTIONID_ANY	**All current network connections**

3.6.5 Network Connection State Values

This is the complete set of permissible values for the TNC_Connection_State type in this version of the IF-IMV API.

Network Connection State Value	Definition
TNC_CONNECTION_STATE_CREATE	Network connection created
TNC_CONNECTION_STATE_HANDSHAKE	Handshake about to start
TNC_CONNECTION_STATE_ACCESS_ALLOWED	Handshake completed. TNCS recommended that requested access be allowed.

TNC_CONNECTION_STATE_ACCESS_ISOLATED	Handshake completed. TNCS recommended that isolated access be allowed.
TNC_CONNECTION_STATE_ACCESS_NONE	Handshake completed. TNCS recommended that no network access be allowed.
TNC_CONNECTION_STATE_DELETE	**About to delete network connection ID. Remove all associated state.**

3.6.6 Handshake Retry Reason Values

This is the complete set of permissible values for the TNC_Retry_Reason type in this version of the IF-IMV API.

Handshake Retry Reason Value	Definition
TNC_RETRY_REASON_IMV_IMPORTANT_POLICY_CHANGE	IMV policy has changed. It recommends handshake retry even if network connectivity must be interrupted
TNC_RETRY_REASON_IMV_MINOR_POLICY_CHANGE	IMV policy has changed. It requests handshake retry but not if network connectivity must be interrupted
TNC_RETRY_REASON_IMV_SERIOUS_EVENT	IMV has detected a serious event and recommends handshake retry even if network connectivity must be interrupted
TNC_RETRY_REASON_IMV_MINOR_EVENT	IMV has detected a minor event. It requests handshake retry but not if network connectivity must be interrupted
TNC_RETRY_REASON_IMV_PERIODIC	IMV wishes to conduct a periodic recheck. It recommends handshake retry but not if network connectivity must be interrupted

3.6.7 IMV Action Recommendation Values

This is the complete set of permissible values for the TNC_IMV_Action_Recommendation type in this version of the IF-IMV API.

IMV Action Recommendation Value	Definition
TNC_IMV_ACTION_RECOMMENDATION_ALLOW	IMV recommends allowing

	access
TNC_IMV_ACTION_RECOMMENDATION_NO_ACCESS	IMV recommends no access
TNC_IMV_ACTION_RECOMMENDATION_ISOLATE	IMV recommends limited access. This access may be expanded after remediation
TNC_IMV_ACTION_RECOMMENDATION_NO_RECOMMENDATION	**IMV does not have a recommendation**

3.6.8 IMV Evaluation Result Values

This is the complete set of permissible values for the TNC_IMV_Evaluation_Result type in this version of the IF-IMV API.

IMV Evaluation Result Value	Definition
TNC_IMV_EVALUATION_RESULT_COMPLIANT	AR complies with policy
TNC_IMV_EVALUATION_RESULT_NONCOMPLIANT_MINOR	AR is not compliant with policy. Non-compliance is minor.
TNC_IMV_EVALUATION_RESULT_NONCOMPLIANT_MAJOR	AR is not compliant with policy. Non-compliance is major.
TNC_IMV_EVALUATION_RESULT_ERROR	IMV is unable to determine policy compliance due to error
TNC_IMV_EVALUATION_RESULT_DONT_KNOW	**IMV does not know whether AR complies with policy**

3.6.9 Vendor ID Values

These are reserved vendor ID values. Other vendor IDs may be used as described in section 3.5.2.9. Note that vendor IDs are assigned by IANA as described in section 3.2.3.

Vendor ID Value	Value	Definition
TNC_VENDORID_TCG	0	Reserved for older TCG-defined values
TNC_VENDORID_TCG_NEW	0x005597	Reserved for newer TCG-defined values
TNC_VENDORID_ANY	**0xffffff**	**Wild card matching any vendor ID**

3.6.10 Message Subtype Values

This is a reserved message subtype value. Other message subtypes may be used as described in section 3.5.2.11. Note that message subtypes are assigned by vendors as described in section 3.5.2.7.

Message Subtype Value	Value	Definition
TNC_SUBTYPE_ANY	**0xff**	**Wild card matching any message**

		subtype

3.6.11 Attribute ID Values and Value Definitions

As described in section 3.5.2.15, attribute IDs with a vendor ID of zero (0) or 0x005597 are reserved for definition by TCG. Some of these reserved attribute IDs are defined in this section. After the table of reserved attribute IDs, a description of the format of the corresponding attribute value is provided.

Attribute ID Value	Value	Meaning
TNC_ATTRIBUTEID_PREFERRED_LANGUAGE	0x00000001	Preferred human-readable language(s)
TNC_ATTRIBUTEID_REASON_STRING	0x00000002	Reason for IMV Recommendation
TNC_ATTRIBUTEID_REASON_LANGUAGE	0x00000003	Language for IMV reason
TNC_ATTRIBUTEID_MAX_ROUND_TRIPS	0x00559700	Maximum number of round trips supported
TNC_ATTRIBUTEID_MAX_MESSAGE_SIZE	0x00559701	Maximum size of message supported
TNC_ATTRIBUTEID_DHPN_VALUE	0x00559702	Unique value for Diffie-Hellman Pre-Negotiation used by IMVs including PTS-IMV
TNC_ATTRIBUTEID_HAS_LONG_TYPES	0x00559703	Flag to indicate whether connection supports long message types
TNC_ATTRIBUTEID_HAS_EXCLUSIVE	0x00559704	Flag to indicate whether connection supports exclusive delivery
TNC_ATTRIBUTEID_HAS_SOH	0x00559705	Flag to indicate whether connection supports SOH
TNC_ATTRIBUTEID_SOH	0x00559706	Contents of SOH

TNC_ATTRIBUTEID_SSOH	0x00559707	Contents of SSOH
TNC_ATTRIBUTEID_IFTNCCS_PROTOCOL	0x0055970A	IF-TNCCS Protocol Name
TNC_ATTRIBUTEID_IFTNCCS_VERSION	0x0055970B	IF-TNCCS Protocol Version
TNC_ATTRIBUTEID_IFT_PROTOCOL	0x0055970C	IF-T Protocol Name
TNC_ATTRIBUTEID_IFT_VERSION	0x0055970D	IF-T Protocol Version
TNC_ATTRIBUTEID_TLS_UNIQUE	0x0055970E	TLS-Unique value used by IMVs including PTS-IMV
TNC_ATTRIBUTEID_PRIMARY_IMV_ID	0x00559710	Primary ID of the IMV
TNC_ATTRIBUTEID_AR_IDENTITIES	0x00559712	AR Identities

3.6.11.1 Preferred Language Attribute

The Preferred Language attribute indicates which human-readable language(s) are preferred for a particular connection or for a TNCS as a whole. An IMV may get the value of the Preferred Language attribute with `TNC_TNCS_GetAttribute` but may not set this value with `TNC_TNCS_SetAttribute`. If the IMV provides a connection ID to `TNC_TNCS_GetAttribute`, the attribute value returned by the TNCS will pertain to that connection. If the IMV provides `TNC_CONNECTIONID_ANY`, the attribute value will pertain to the TNCS as a whole. A TNCS may support only one of these options, or it may support both.

The attribute value for the Preferred Language attribute is a NUL-terminated UTF-8 string containing an Accept-Language header as defined in IETF RFC 3282 [3] (US-ASCII only, no control characters allowed). This header lists the languages preferred for human-readable messages. The TNCS may obtain information about language preferences through IF-TNCCS or in some other manner. If no language preference information is available, a zero length string is used (although the string actually contains one byte, the NUL terminator). An IMV must be able to handle this case, which may be common if a TNCS supports this function but the TNCC does not provide language preference information. Note that the byte length of the Preferred Language attribute always includes the NUL terminator. In fact, it includes every byte in the buffer.

TNCSs are not required to implement the Preferred Language attribute but they SHOULD do so if possible since this feature helps with internationalization and results in a better user experience. Likewise, IMVs SHOULD make use of this function when available but are not required to do so. Many TNCSs do not support this attribute. IMVs MUST work properly if a TNCS does not support it.

3.6.11.2 Reason String Attribute

The Reason String attribute allows an IMV to deliver to the TNCS a reason string explaining its IMV Action Recommendation and IMV Evaluation Result. The TNCS MAY pass the reason string to the TNC Client via IF-TNCCS and either the TNCS or the TNCC MAY log the string, modify it, ignore it, combine it with other strings, or take another action with it (as long as they meet the requirements of this specification).

An IMV may set the value of the Reason String attribute with `TNC_TNCS_SetAttribute` but may not get this value with `TNC_TNCS_GetAttribute`. The IMV MUST provide a valid connection ID when setting this attribute.

The attribute value for the Reason String attribute is a NUL-terminated UTF-8 string which explains the reason for the IMV's IMV Action Recommendation and IMV Evaluation Result. In constructing the reason string, the IMV SHOULD try to accommodate the language preferences conveyed via the `TNC_TNCS_GetAttribute` function with AttributeID set to `TNC_ATTRIBUTEID_REASON_LANGUAGE`. No standard format for the reason string has been defined. Any format is permissible and MUST be accommodated by the TNCS. However, the TNCS or TNCC MAY modify or ignore the reason string, as noted above.

A zero length string is permissible, although it is generally not necessary since this is semantically equivalent to not setting a value for the Reason String attribute. Note that a zero length reason string actually contains one byte, the NUL terminator. Thus the value for this attribute MUST never have a byte length of zero.

Many TNCSs do not support this attribute. IMVs MUST work properly if a TNCS does not support it.

An IMV that does want to provide a reason string SHOULD do so before providing an IMV Action Recommendation since the TNCS may decide to terminate the handshake immediately based on the IMV Action Recommendation.

3.6.11.3 Reason Language Attribute

The Reason Language attribute allows an IMV to indicate to the TNCS which language or languages were used in a reason string. The TNCS MAY pass this language information along with reason string to the TNC Client via IF-TNCCS and either the TNCS or the TNCC MAY log this information, modify it, ignore it, combine it with other language information, or take another action with it (as long as they meet the requirements of this specification).

An IMV may set the value of the Reason Language attribute with `TNC_TNCS_SetAttribute` but may not get this value with `TNC_TNCS_GetAttribute`. The IMV MUST provide a valid connection ID when setting this attribute.

The attribute value for the Reason Language attribute is a NUL-terminated UTF-8 string containing an RFC 3066 language tag. A zero length reason language string (one with only a NUL terminator) is permissible, although it is generally not necessary since this is semantically equivalent to not setting a value for the Reason Language attribute. Note that a zero length reason language string actually contains one byte, the NUL terminator. Thus the value for this attribute MUST never have a byte length of zero.

Many TNCSs do not support this attribute. IMVs MUST work properly if a TNCS does not support it. Likewise, TNCSs MUST work properly if an IMV does not set this attribute.

An IMV that does want to provide a reason language string SHOULD do so before providing an IMV Action Recommendation since the TNCS may decide to terminate the handshake immediately based on the IMV Action Recommendation.

3.6.11.4 Maximum Round Trips Attribute

The Maximum Round Trips attribute allows a TNCS to indicate to the IMVs the maximum number of round trips the underlying transport supports. An IMV may get the value of the Maximum Round Trips attribute with `TNC_TNCS_GetAttribute` but may not set this value with `TNC_TNCS_SetAttribute`. The IMV MUST provide a valid connection ID to `TNC_TNCS_GetAttribute`. The attribute value returned by the TNCS will pertain to that connection.

The attribute value for Maximum Round Trips is a 4 byte unsigned integer (most significant byte first) representing the maximum number of round trips allowed by the underlying transport. A value of zero (0) indicates that the maximum number of round trips for this connection is

unknown, and a value of 0xffffffff indicates that the number of round trips is unlimited. The length for this attribute MUST always be 4.

TNCSs are not required to implement the Maximum Round Trips attribute, but they SHOULD do so if possible. Likewise, IMVs SHOULD make use of this function when available but are not required to do so. In addition, the TNCS SHOULD provide the Action Recommendation to the TNCC on or before the last round trip. If the Maximum Round Trips is exceeded, the TNCS MUST return the TNC_RESULT_EXCEEDED_MAX_ROUND_TRIPS result code from the TNC_TNCS_SendMessage function. Many TNCSs do not support this attribute; IMVs MUST work properly if a TNCS does not support it.

3.6.11.5 Maximum Message Size Attribute

The Maximum Message Size attribute allows a TNCS to indicate to the IMVs the maximum message size the underlying transport supports. An IMV may get the value of the Maximum Message Size attribute with `TNC_TNCS_GetAttribute` but may not set this value with `TNC_TNCS_SetAttribute`. The IMV MUST provide a valid connection ID to `TNC_TNCS_GetAttribute`. The attribute value returned by the TNCS will pertain to that connection.

The attribute value for Maximum Message Size is a 4 byte unsigned integer (most significant byte first) representing the maximum message size allowed by the underlying transport. The maximum message size should be calculated to be the maximum number of bytes that a single IMV should send in a single batch (combining the data in all messages sent by that IMV in that batch). A value of zero (0) indicates that the maximum message size is unknown, and a value of 0xffffffff indicates that the message size is unlimited. The length for this attribute MUST always be 4.

TNCSs are not required to implement the Maximum Message Size attribute, but they SHOULD do so if possible. Likewise, IMVs SHOULD make use of this function when available but are not required to do so. If the Maximum Message Size is exceeded, the TNCS MUST return the TNC_RESULT_EXCEEDED_MAX_MESSAGE_SIZE result code. Many TNCSs do not support this attribute; IMVs MUST work properly if a TNCS does not support it.

3.6.11.6 Diffie-Hellman Pre-Negotiation (DHPN) Value Attribute

The Diffie-Hellman Pre-Negotiation (DHPN) attribute allows a TNCS to provide a Unique-Value-1 (as described in IF-T for Tunneled EAP Methods) to the PTS-IMV, so that this value can be used for verifying the PTS-IMC's Integrity Report. Other IMVs may also use this value as well. An IMV may get the value of the DHPN Value attribute with `TNC_TNCS_GetAttribute` but may not set this value with `TNC_TNCS_SetAttribute`. The IMV MUST provide a valid connection ID to `TNC_TNCS_GetAttribute`. The attribute value returned by the TNCS will pertain to that connection.

The attribute value for DHPN is a 20 byte value representing the Unique-Value-1 provided by the underlying transport. The length for this attribute MUST always be 20.

TNCSs are not required to implement the DHPN attribute, but they SHOULD do so if possible. Likewise, IMVs SHOULD make use of this function when available but are not required to do so. Many TNCSs do not support this attribute; IMVs MUST work properly if a TNCS does not support it.

3.6.11.7 Has Long Types Attribute

The Has Long Types attribute allows a TNCS to indicate to the IMVs that a particular connection supports long message types. An IMV may get the value of the Has Long Types attribute with `TNC_TNCS_GetAttribute` but may not set this value with `TNC_TNCS_SetAttribute`. The IMV MUST provide a valid connection ID to `TNC_TNCS_GetAttribute`. The attribute value returned by the TNCS will pertain to that connection. If the IMV does not provide a valid connection ID, the TNCV SHOULD return `TNC_RESULT_INVALID_PARAMETER`, since some connections may support long types and others may not.

The attribute value for Has Long Types is one byte with a value of zero (0) if the connection does not support long message types, and a value of one (1) if the connection does support long message types. The length for this attribute MUST always be 1. If the value of this attribute is 1, then an IMV may call the `TNC_TNCS_SendMessageLong` function for this connection and should expect to receive calls for this connection to the IMV's `TNC_IMV_ReceiveMessageLong` function (if implemented by the IMV).

TNCSs MUST implement the Has Long Types attribute if they support the `TNC_TNCS_SendMessageLong` and `TNC_IMV_ReceiveMessageLong` functions. IMVs MAY make use of this attribute when available but are not required to do so. Many TNCSs do not support this attribute; IMVs MUST work properly if a TNCS does not support it.

3.6.11.8 Has Exclusive Attribute

The Has Exclusive attribute allows a TNCS to indicate to the IMVs that a particular connection supports exclusive delivery. An IMV may get the value of the Has Exclusive attribute with `TNC_TNCS_GetAttribute` but may not set this value with `TNC_TNCS_SetAttribute`. The IMV MUST provide a valid connection ID to `TNC_TNCS_GetAttribute`. The attribute value returned by the TNCS will pertain to that connection. If the IMV does not provide a valid connection ID, the TNCS SHOULD return `TNC_RESULT_INVALID_PARAMETER`, since some connections may support exclusive delivery and others may not.

The attribute value for Has Exclusive is one byte with a value of zero (0) if the connection does not support exclusive delivery, and a value of one (1) if the connection does support exclusive delivery. The length for this attribute MUST always be 1. If the value of this attribute is 1, then an IMV may call the `TNC_TNCS_SendMessageLong` function for this connection and set the `TNC_MESSAGE_FLAGS_EXCLUSIVE` flag.

TNCSs MUST implement the Has Exclusive attribute if they support the `TNC_TNCS_SendMessageLong` and `TNC_IMV_ReceiveMessageLong` functions because the use of this attribute is essential to the proper use of those functions. However, some or even all connections for those TNCSs may have a value of 0 for this attribute, depending on the IF-TNCCS protocol in use. IMVs MAY make use of this attribute when available but are not required to do so. Many TNCSs do not support this attribute; IMVs MUST work properly if a TNCS does not support it.

3.6.11.9 Has SOH Attribute

The Has SOH attribute allows a TNCS to indicate to the IMVs that a particular connection supports SOH functions like `TNC_TNCS_SendMessageSOH`. An IMV may get the value of the Has SOH attribute with `TNC_TNCS_GetAttribute` but may not set this value with `TNC_TNCS_SetAttribute`. The IMV MUST provide a valid connection ID to `TNC_TNCS_GetAttribute`. The attribute value returned by the TNCS will pertain to that connection. If the IMV does not provide a valid connection ID, the TNCS SHOULD return `TNC_RESULT_INVALID_PARAMETER`, since some connections may support SOH and others may not.

The attribute value for Has SOH is one byte with a value of zero (0) if the connection does not support SOH, and a value of one (1) if the connection does support SOH. The length for this attribute MUST always be 1.

If the value of the Has SOH attribute is 1, then the TNCS MUST implement the SOH and SSOH attributes and the `TNC_TNCS_SendMessageSOH` function. Further, the TNCS MUST be prepared to call an IMV's `TNC_IMV_ReceiveMessageSOH` function to deliver messages if the IMV implements this function. An IMV may therefore call the `TNC_TNCS_SendMessageSOH` function for this connection and should expect to receive calls for this connection to the IMV's `TNC_IMV_ReceiveMessageSOH` function (if implemented by the IMV). If an IMV does not implement `TNC_IMV_ReceiveMessageSOH` and an IF-TNCCS-SOH connection is started,

messages will be delivered to that IMV using `TNC_IMV_ReceiveMessage` instead, as described in section 3.8.5.

A TNCS MUST implement the Has SOH attribute if it supports the `TNC_TNCS_SendMessageSOH` and `TNC_IMV_ReceiveMessageSOH` functions. IMVs MAY use the Has SOH attribute when available but are not required to do so. Many TNCSs do not support this attribute; IMVs MUST work properly if a TNCS does not support it.

3.6.11.10 SOH Attribute

The SOH attribute allows IMVs to obtain a copy of the full SOH that was received from the TNCS on a particular connection. An IMV may get the value of the SOH attribute with `TNC_TNCS_GetAttribute` but may not set this value with `TNC_TNCS_SetAttribute`. The IMV MUST provide a valid connection ID to `TNC_TNCS_GetAttribute`, and this connection must support SOH, indicating this by having a 1 value for the Has SOH attribute. Further, the TNCC must have sent a message to the TNCS on this connection (as would be the case if the IMV receives a message on this connection or if the TNCS calls `TNC_IMV_BatchEnding` on this connection). If any of these requirements is not met, the TNCS SHOULD return `TNC_RESULT_INVALID_PARAMETER`, since no SOH is yet available.

The attribute value for the SOH attribute is a sequence of bytes that contains the binary value of the SoH sent by the TNCC on this connection, as described in section 3.6.1.1 of [12]. No padding or encapsulation should be added, so the first byte in the value is the first byte in the SoH (the first byte in the SoH's header), and the last byte is the last byte in the SoH (the last byte in the last SoHReportEntry). The length for this attribute will vary depending on its contents.

A TNCS MUST implement the SOH attribute for a connection if the value of the Has SOH attribute for that connection is 1. IMVs MAY make use of this attribute when available but are not required to do so. Many TNCSs do not support this attribute; IMVs MUST work properly if a TNCS does not support it.

3.6.11.11 SSOH Attribute

The SSOH attribute allows IMVs to obtain a copy of the SSOH that was received from the TNCS on a particular connection. An IMV may get the value of the SSOH attribute with `TNC_TNCS_GetAttribute` but may not set this value with `TNC_TNCS_SetAttribute`. The IMV MUST provide a valid connection ID to `TNC_TNCS_GetAttribute`, and this connection must support SOH, indicating this by having a 1 value for the Has SOH attribute. Further, the TNCC must have sent a message to the TNCS on this connection (as would be the case if the IMV receives a message on this connection or if the TNCS calls `TNC_IMV_BatchEnding` on this connection). If any of these requirements is not met, the TNCS SHOULD return `TNC_RESULT_INVALID_PARAMETER`, since no SSOH is yet available.

The attribute value for the SSOH attribute is a sequence of bytes that contains the binary value of the SSoH field in the SoH sent by the TNCS on this connection, as described in section 3.6.1.2 of [12]. No padding or encapsulation should be added, so the first byte in the value is that first byte in the SSoH (the first byte in the SSoH's System-Health-Id attribute), and the last byte is the last byte in the SSoH (the last byte in the Vendor-Specific attribute that contains the SSoH attributes). The length for this attribute will vary depending on its contents.

A TNCS MUST implement the SSOH attribute for a connection if the value of the Has SOH attribute for that connection is 1. IMVs MAY make use of this attribute when available but are not required to do so. Many TNCSs do not support this attribute; IMVs MUST work properly if a TNCS does not support it.

3.6.11.12 IF-TNCCS Protocol Attribute

The IF-TNCCS Protocol attribute allows IMVs to determine which IF-TNCCS Protocol or similar protocol is being used for a particular connection. An IMV may get the value of the IF-TNCCS Protocol attribute with `TNC_TNCS_GetAttribute` but may not set this value with `TNC_TNCS_SetAttribute`. The IMV MUST provide a valid connection ID to

TNC_TNCS_GetAttribute. The attribute value returned by the TNCS will pertain to that connection. If the IMV does not provide a valid connection ID, the TNCS SHOULD return TNC_RESULT_INVALID_PARAMETER, since different connections may use different IF-TNCCS protocols.

The attribute value for the IF-TNCCS Protocol attribute is a NUL-terminated UTF-8 string containing the standard name for the IF-TNCCS protocol in use for the specified connection. There are two standard values for this attribute at this time: "IF-TNCCS" for connections that use IF-TNCCS 1.0 [15], IF-TNCCS 1.1 [16], or IF-TNCCS 2.0 [14], and "IF-TNCCS-SOH" for connections that use IF-TNCCS-SOH 1.0 [15]. In both cases, only ASCII characters are included in these names (and no control characters except for the terminating NUL). The byte length of the IF-TNCCS Protocol attribute always includes the NUL terminator. In fact, it includes every byte in the buffer.

Although the two strings listed above are the only values defined for this attribute in this specification, TNCSs are not restricted from returning other values if the IF-TNCCS protocol is not one of the ones specified above. Future IF-TNCCS protocol specifications should define which string will appear in this attribute, and future versions of this specification should list the strings included in these specifications. Vendors may define their own IF-TNCCS Protocol strings that can be contained in this attribute. If they do so, these strings MUST start with the vendor's SMI Private Enterprise Number expressed in ASCII as a decimal integer with no leading zeros, followed by a space (U+0020). The rest of the string can be any UTF-8 encoded Unicode characters, except that NUL (U+0000) must only appear as the last character in the string. Future IF-TNCCS protocols may also relax the ASCII-only convention for these strings (although they are encouraged not to), so IMVs should be prepared to receive non-ASCII characters. Despite the flexibility described in this paragraph, TNCS vendors are encouraged to restrict themselves to the standardized IF-TNCCS protocols as much as possible.

If the TNCS cannot return information about which IF-TNCCS Protocol is used for a particular connection, or if no IF-TNCCS Protocol or other similar protocol is used, the TNCS SHOULD return TNC_RESULT_INVALID_PARAMETER. The IMV may wish to check back later in the lifecycle of the connection since this information might be available later. In spite of this provision, TNCSs SHOULD have this information available for all connections before calling any IMV functions with that connection ID. TNCSs SHOULD NOT change the IF-TNCCS Protocol information for a connection during the life of that connection but may do so if a connection ID is reused (i.e. if the connection state changes to TNC_CONNECTION_STATE_DELETE, then the connection ID is used again, and the connection state changes to TNC_CONNECTION_STATE_CREATE).

TNCSs SHOULD implement the IF-TNCCS Protocol attribute for all connections. IMVs MAY make use of this attribute when available but are not required to do so. Many TNCSs do not support this attribute; IMVs MUST work properly if a TNCS does not support it.

3.6.11.13 IF-TNCCS Version Attribute

The IF-TNCCS Version attribute allows IMVs to determine the version of the IF-TNCCS Protocol or similar protocol that is being used for a particular connection. An IMV may get the value of the IF-TNCCS Version attribute with TNC_TNCS_GetAttribute but may not set this value with TNC_TNCS_SetAttribute. The IMV MUST provide a valid connection ID to TNC_TNCS_GetAttribute. The attribute value returned by the TNCS will pertain to that connection. If the IMV does not provide a valid connection ID, the TNCS SHOULD return TNC_RESULT_INVALID_PARAMETER, since different connections may use different IF-TNCCS protocols with different versions.

The attribute value for the IF-TNCCS Protocol attribute is a NUL-terminated UTF-8 string containing the version of the IF-TNCCS protocol in use for the specified connection. There are three standard values for this attribute at this time: "1.0" for connections that use IF-TNCCS 1.0 [15] or IF-TNCCS-SOH 1.0 [12], "1.1" for connections that use IF-TNCCS 1.1 [16], and "2.0" for connections that use IF-TNCCS 2.0 [14]. In all cases, only ASCII characters are included in these

names (and no control characters except for the terminating NUL). The byte length of the IF-TNCCS Protocol attribute always includes the NUL terminator. In fact, it includes every byte in the buffer.

Although the three strings listed above are the only values defined for this attribute in this specification, TNCSs are not restricted from returning other values if the IF-TNCCS protocol version is not one of the ones specified above. Future IF-TNCCS protocol specifications may define which string will appear in this attribute, and future versions of this specification may list the strings included in these specifications. If possible, the version format should be similar to the one used above. However, this may not be possible, so IMVs should be prepared to receive any NUL-terminated UTF-8 string. Vendors may define their own IF-TNCCS Version strings that can be contained in this attribute. These strings can contain any UTF-8 encoded Unicode characters, except that NUL (U+0000) must only appear as the last character in the string.

If the TNCS cannot return information about which IF-TNCCS protocol version is used for a particular connection, or if no IF-TNCCS Protocol or other similar protocol is used, the TNCS SHOULD return `TNC_RESULT_INVALID_PARAMETER`. The IMV may wish to check back later in the lifecycle of the connection, since this information might be available later. In spite of this provision, TNCS's SHOULD have this information available for all connections before calling any IMV functions with that connection ID. TNCSs SHOULD NOT change the IF-TNCCS protocol version information for a connection during the life of that connection but may do so if a connection ID is reused (i.e. if the connection state changes to `TNC_CONNECTION_STATE_DELETE`, then the connection ID is used again, and the connection state changes to `TNC_CONNECTION_STATE_CREATE`).

TNCSs SHOULD implement the IF-TNCCS Version attribute for all connections. IMVs MAY make use of this attribute when available but are not required to do so. Many TNCSs do not support this attribute; IMVs MUST work properly if a TNCS does not support it.

3.6.11.14 IF-T Protocol Attribute

The IF-T Protocol attribute allows IMVs to determine which IF-T Protocol or other transport protocol is being used for a particular connection. An IMV may get the value of the IF-T Protocol attribute with `TNC_TNCS_GetAttribute` but may not set this value with `TNC_TNCS_SetAttribute`. The IMV MUST provide a valid connection ID to `TNC_TNCS_GetAttribute`. The attribute value returned by the TNCS will pertain to that connection. If the IMV does not provide a valid connection ID, the TNCS SHOULD return `TNC_RESULT_INVALID_PARAMETER`, since different connections may use different IF-TNCCS protocols.

The attribute value for the IF-T Protocol attribute is a NUL-terminated UTF-8 string containing the standard name for the IF-T protocol in use for the specified connection. There are three standard values for this attribute at this time: two of them are "IF-T for Tunneled EAP" for connections that use IF-T for Tunneled EAP Methods 1.0 [10] or IF-T for Tunneled EAP Methods 1.1 [11] or IF-T for Tunneled EAP Methods, “IF-T for TLS” for connections that use IF-T Binding to TLS 1.0 [19] or IF-T Binding to TLS 2.0 [24]. In addition, connections that use PEAP [17] as a transport should use "PEAP" as the string to be returned in this attribute. In both cases, only ASCII characters are included in these names (and no control characters except for the terminating NUL). The byte length of the IF-T Protocol attribute always includes the NUL terminator. In fact, it includes every byte in the buffer.

Although the three strings listed above are the only values defined for this attribute in this specification, TNCSs are not restricted from returning other values if the IF-T protocol is not one of the ones specified above. Future IF-T protocol specifications may define which string will appear in this attribute, and future versions of this specification may list the strings included in these specifications. Vendors may define their own IF-T Protocol strings that can be contained in this attribute. If they do so, these strings should start with the vendor's SMI Private Enterprise Number expressed in ASCII as a decimal integer with no leading zeros, followed by a space (U+0020). The rest of the string can be any UTF-8 encoded Unicode characters except that NUL

(U+0000) must only appear as the last character in the string. Future IF-T protocols may also relax the ASCII-only convention for these strings (although they are encouraged not to), so IMVs should be prepared to receive non-ASCII characters. Despite the flexibility described in this paragraph, TNCS vendors are encouraged to restrict themselves to the standardized IF-T protocols as much as possible.

If the TNCS cannot return information about which IF-T Protocol is used for a particular connection, or if no IF-T Protocol or other transport protocol is used, the TNCS SHOULD return `TNC_RESULT_INVALID_PARAMETER`. The IMV may wish to check back later in the lifecycle of the connection, since this information might be available later. In spite of this provision, TNCSs SHOULD have this information available for all connections before calling any IMV functions with that connection ID. TNCSs SHOULD NOT change the IF-T Protocol information for a connection during the life of that connection but may do so if a connection ID is reused (i.e. if the connection state changes to `TNC_CONNECTION_STATE_DELETE`, then the connection ID is used again, and the connection state changes to `TNC_CONNECTION_STATE_CREATE`).

TNCSs SHOULD implement the IF-T Protocol attribute for all connections. IMVs MAY make use of this attribute when available but are not required to do so. Many TNCSs do not support this attribute; IMVs MUST work properly if a TNCS does not support it.

3.6.11.15 IF-T Version Attribute

The IF-T Version attribute allows IMVs to determine the version of the IF-T Protocol or other transport protocol that is being used for a particular connection. An IMV may get the value of the IF-T Version attribute with `TNC_TNCS_GetAttribute` but may not set this value with `TNC_TNCS_SetAttribute`. The IMV MUST provide a valid connection ID to `TNC_TNCS_GetAttribute`. The attribute value returned by the TNCS will pertain to that connection. If the IMV does not provide a valid connection ID, the TNCS SHOULD return `TNC_RESULT_INVALID_PARAMETER`, since different connections may use different IF-T protocols with different versions.

The attribute value for the IF-T Protocol attribute is a NUL-terminated UTF-8 string containing the version of the IF-T protocol in use for the specified connection. There are three standard values for this attribute at this time: "1.0" for connections that use IF-T for Tunneled EAP Methods 1.0 or IF-T Binding to TLS 1.0 [19], "1.1" for connections that use IF-T for Tunneled EAP Methods 1.1 [11], and "2.0" for connections that use IF-T for Tunneled EAP Methods 2.0 [24]. In all cases, only ASCII characters are included in these names (and no control characters except for the terminating NUL). The byte length of the IF-T Protocol attribute always includes the NUL terminator. In fact, it includes every byte in the buffer.

Although the strings listed above are the only values defined for this attribute in this specification, TNCSs are not restricted from returning other values if the IF-T protocol version is not one of the ones specified above. In fact, complex transport protocol stacks may result in complex version information such as the version of IF-T for Tunneled EAP Methods, the version of the tunneled EAP method (e.g. EAP-TTLS), and the version of an underlying carrier protocol (e.g. TLS 1.2). Future IF-T protocol specifications should define which string will appear in this attribute and future versions of this specification should list the strings included in these specifications. If possible, the version format should be similar to the one used above. However, this may not be possible, so IMVs should be prepared to receive any NUL-terminated UTF-8 string. Vendors may define their own IF-T Version strings that can be contained in this attribute. These strings can contain any UTF-8 encoded Unicode characters, except that NUL (U+0000) must only appear as the last character in the string.

If the TNCS cannot return information about which IF-T protocol version is used for a particular connection, or if no IF-T Protocol or other transport protocol is used, the TNCS SHOULD return `TNC_RESULT_INVALID_PARAMETER`. The IMV may wish to check back later in the lifecycle of the connection, since this information might be available later. In spite of this provision, TNCS's SHOULD have this information available for all connections before calling any IMV functions with that connection ID. TNCSs SHOULD NOT change the IF-T protocol version information for a

connection during the life of that connection but may do so if a connection ID is reused (i.e. if the connection state changes to TNC_CONNECTION_STATE_DELETE, then the connection ID is used again, and the connection state changes to TNC_CONNECTION_STATE_CREATE).

TNCSs SHOULD implement the IF-T Version attribute for all connections. IMVs MAY make use of this attribute when available but are not required to do so. Many TNCSs do not support this attribute; IMVs MUST work properly if a TNCS does not support it.

3.6.11.16 Primary IMV ID Attribute

The Primary IMV ID attribute indicates the unique identifier of the IMV assigned by the TNCS when the TNCS loads this IMV. An IMV may get the value of the Primary IMV ID attribute with TNC_TNCS_GetAttribute but may not set this value with TNC_TNCS_SetAttribute.

The attribute value for this attribute is TNC_IMVID.

TNCSs MUST implement the Primary IMV ID attribute if they support the TNC_TNCS_SendMessageLong and TNC_IMV_ReceiveMessageLong functions, because the use of this attribute is essential to the proper use of those functions. IMVs MAY make use of this attribute when available but are not required to do so. Many TNCSs do not support this attribute; IMVs MUST work properly if a TNCS does not support it. IMVs that use the DLL binding will generally not need this attribute, since the TNCS passes the primary IMV ID to the IMV when it calls the TNC_IMV_Initialize function. The main purpose of this function is for IMVs that use the Java binding and support multiple IMV IDs, since the TNCS does not generally pass the primary IMV ID to the IMV with the Java binding.

3.6.11.17 TLS-Unique Attribute

The TLS-Unique attribute allows a TNCS that supports TLS-Unique (as described in IF-T for Tunneled EAP Methods) to provide to a PTS-IMV, or other IMVs, a TLS-Unique value obtained from the underlying TLS session, so that this value can be used in the TPM_Quote operation. An IMV may get the value of the TLS-Unique value with TNC_TNCS_GetAttribute but may not set this value with TNC_TNCS_SetAttribute. The IMV MUST provide a valid connection ID to TNC_TNCS_GetAttribute. The attribute value returned by the TNCS will pertain to that connection.

The attribute value for TLS-Unique is a byte array representing the tls-unique value provided by the underlying TLS transport. The length for this attribute may vary, depending on the cipher suite used. For many cipher suites, the length will be 12 octets.

TNCSs are not required to implement the TLS-Unique attribute, but they SHOULD do so if possible. IMVs MAY make use of this attribute when available but are not required to do so. Many TNCSs do not support this attribute; IMVs MUST work properly if a TNCS does not support it.

3.6.11.18 AR Identities Attribute

The AR Identities attribute provides a list of identities that the AR established by authenticating to the TNCS or NAA. An IMV may get the value of the AR Identities attribute with TNC_TNCS_GetAttribute but may not set this value with TNC_TNCS_SetAttribute. The IMV MUST provide a valid connection ID to TNC_TNCS_GetAttribute. The attribute value returned by the TNCS will pertain to that connection.

This information can be used for many purposes, such as correlating current integrity measurements with previous measurements from the same AR or using the identity to help determine which sets of policies should apply.

TNCSs are not required to implement the AR Identities attribute but they SHOULD do so if possible since this feature helps IMVs to personalize their behavior based on user and machine identity. If a TNCS does implement this attribute, it MUST use the attribute value format described below. IMVs MAY make use of this attribute when available but MUST work properly if a TNCS does not support it.

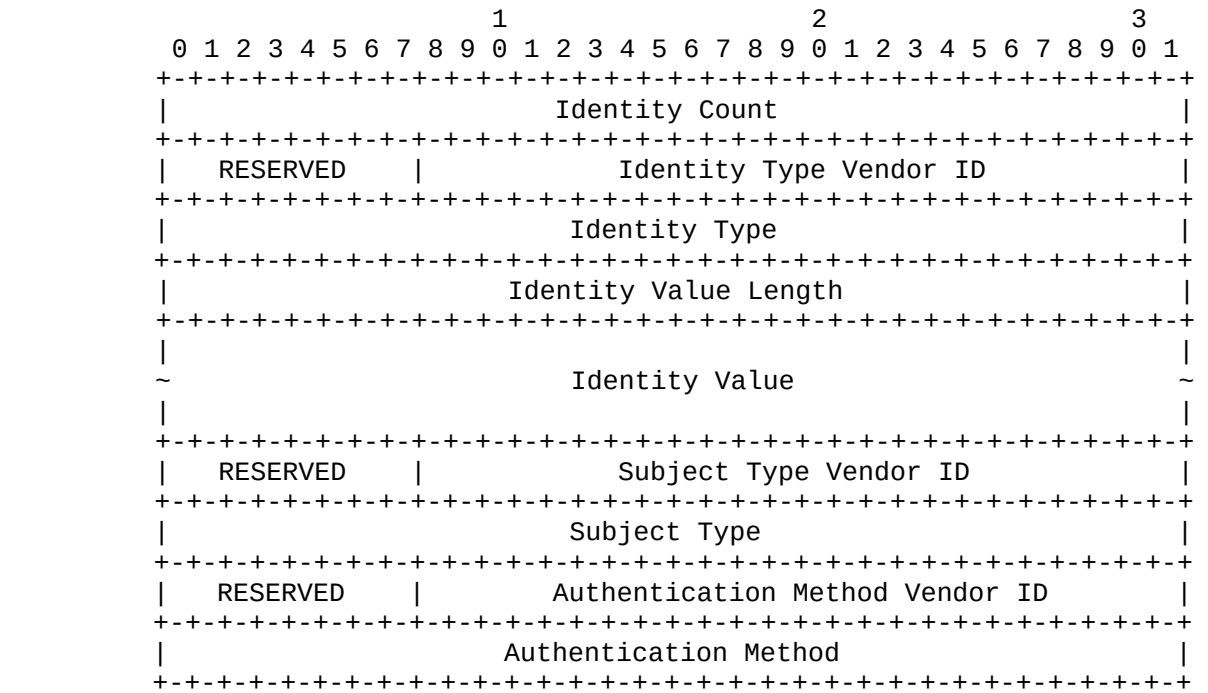
The attribute value for the AR Identities attribute is a binary data structure containing a list of identities that the AR has established. The number of identities in the list may be any integer greater than or equal to zero (since an AR may establish zero or more machine identities and zero or more user identities). Each identity is composed of four components: identity type, identity value, subject type, and authentication method. This section defines the syntax of the AR Identities attribute using diagrams. Each diagram depicts the format and size of each field in bits. Implementations MUST send the bits in each diagram as they are shown, traversing the diagram from top to bottom and then from left to right within each line (which represents a 32-bit quantity). Multi-byte fields representing numeric values MUST be sent in network (big endian) byte order, regardless of the native endian format of the processor on which the code is executing. Descriptions of bit field values (e.g. flags) are described referring to the position of the bit within the field. These bit positions are numbered from the most significant bit through the least significant bit so a one byte field with only bit 0 set has the value 0x80.

Note that two ARs may establish the same identity. For example, a particular user may log in from their phone and their laptop at the same time. Even machine identity may not be unique to a single machine. For example, two web servers may be configured with the same DNS name for load balancing purposes. With virtual machines (VMs), it's especially easy to end up with duplicate identities – just start up two copies of the same VM image. If unique machine identities are desired, extra effort must be expended to ensure that uniqueness is established and maintained.

3.6.11.18.1 Attribute Format

Here is the overall format of the AR Identities attribute.

Note that the length of this attribute MUST NOT be zero. If the TNCS does not recognize this attribute ID or does not have any information in this attribute about whether an AR identity was established, the `TNC_TNCS_GetAttribute` function SHOULD return `TNC_RESULT_INVALID_PARAMETER`. This means that no information about AR identities is available via this mechanism. But if the `TNC_TNCS_GetAttribute` function returns `TNC_RESULT_SUCCESS`, the attribute value MUST contain the structure defined below.



Here is a description of the fields in the attribute.

Identity Count (4 octets) - Indicates the number of identities to follow (which can be zero). For each such identity, the attribute MUST contain one each of the remaining fields shown in the diagram above. Although the number of identities in an AR Identities attribute could be calculated by parsing the entire attribute, it is helpful to have this value at the start so that storage for the identities can be allocated prior to parsing. If this number is zero, the AR did not establish any identities.

RESERVED – These fields MUST be set to zero by the TNC Server and MUST be ignored by the IMV.

Identity Type Vendor ID (3 octets) – The SMI Private Enterprise Number associated with the immediately following Identity Type field. If this value is 0x005597 (TCG's SMI PEN), the Identity Type is one of the values listed in section 3.6.11.18.2. For other Identity Type Vendor ID values, the meaning of the Identity Type field is assigned by the vendor whose SMI Private Enterprise Number matches the value in this field.

Identity Type (4 octets) – The type of the identity value, for example, X.500 distinguished name or email address. This field is qualified by the Identity Type Vendor ID field.

Identity Value Length (4 octets) – The length in octets of the immediately following Identity Value field.

Identity Value (variable length) – An AR identity (identifier) of the type indicated by the Identity Type Vendor ID and Identity Type fields. The Identity Value field contains a human-readable UTF-8 string. The format and semantics of this field depends on the Identity Type. The Identity Value field MUST be sized to fit the AR identity string and MUST NOT include extra octets for padding or NUL character termination.

Subject Type Vendor ID (3 octets) - The SMI Private Enterprise Number associated with the immediately following Subject Type field. If this value is 0x005597, the Subject Type is one of the values listed in section 3.6.11.18.3. For other Subject Type Vendor ID values, the meaning of the Subject Type field is assigned by the vendor whose SMI Private Enterprise Number matches the value in this field.

Subject Type (4 octets) – The type of subject identified by the Identity Value, for example, machine or user. This field is qualified by the Subject Type Vendor ID field.

Authentication Method Vendor ID (3 octets) – The SMI Private Enterprise Number associated with the immediately following Authentication Method field. If this value is 0x005597, the Authentication Method is one of the values listed in section 3.6.11.18.4. For other Authentication Method Vendor ID values, the meaning of the Authentication Method field is assigned by the vendor whose SMI Private Enterprise Number matches the value in this field.

Authentication Method (4 octets) – The authentication method used to establish the Identity Value, for example, certificate-based authentication. This field is qualified by the Authentication Method Vendor ID field.

3.6.11.18.2TCG Standard Identity Types

These identity types MUST be used with an Identity Type Vendor ID of 0x005597. The Identity Types are modeled after the IKEv2 ID Types as defined in section 3.5 of RFC 5996 but whereas the IKEv2 Identification Payload uses a binary encoding of its Identification Data field, a UTF-8 string representation is used for the Identity Value field of TCG Standard Identity Types.

Identity Type	Identity Value
0 (TNC_ID_UNKNOWN)	The type of the Identity Value is unknown, perhaps because it was lost due to inadequate APIs. The Identity Value is an opaque UTF-8 string the content

	of which cannot be interpreted but is printable.
1 (TNC_ID_IPV4_ADDR)	A single IPv4 address. The IPv4 address MUST be in dot-decimal notation (i.e. dotted quad notation) consisting of four dot-separated decimal numbers between 0 and 255. No leading 0s are allowed except that the number 0 is represented by a single 0 character. All characters in the TNC_ID_IPV4_ADDR are ASCII and consist of the numbers 0…9 and the separation character ".". An example of a TNC_ID_IPV4_ADDR is "198.51.100.2".
2 (TNC_ID_IPV6_ADDR)	A single IPv6 address. The IPv6 address MUST have the form x:x:x:x:x:x:x:x, where the 'x's are the lowercase hexadecimal values of the eight 16-bit pieces of the address. No leading zeros are allowed except that the number 0 is represented by a single 0 character. All characters in the TNC_ID_IPV6_ADDR MUST be ASCII and consist of the numbers 0…9, the lower case characters a...f and the separation character ":". Examples of a TNC_ID_IPV6_ADDR are “2001:db8:7654:3210:fedc:ba98:7654:3210” or “2001:db8:0:0:8:800:200c:417a”.
3 (TNC_ID_FQDN)	A fully-qualified domain name string. All characters in the TNC_ID_FQDN MUST be ASCII; for an "internationalized domain name", the syntax MUST be as defined in the IDNA2008 documents [5]. An example of a TNC_ID_FQDN is "example.com".
4 (TNC_ID_EMAIL_ADDR)	An email address. This string MUST be an SMTPUTF8 email address, as defined in [6]. This means that the email address may be an all-ASCII address or it may include non-ASCII characters. IMVs MUST treat the Identity Value field as a UTF-8 string, not as pure ASCII. An example of a TNC_ID_EMAIL_ADDR is "joe@example.com". Another example (which includes a non-ASCII character) is "josé@example.com".
5 (TNC_ID_USERNAME)	An UTF-8 string containing a user name. There are no restrictions on the contents of this field, other than the fact that it MUST be a UTF-8 string. An example of a TNC_ID_USERNAME is "John Smith".
6 (TNC_ID_ X500_DN)	An X.500 Distinguished Name. This string MUST be formatted as a UTF-8 string using the algorithm defined in [7]. For AttributeTypes not listed in the table on page 4 of that specification, the dotted-decimal notation MUST be used. An example of a TNC_ID_ X500_DN is “CN= John Smith, O=ACME, C=US”.

3.6.11.18.3TCG Standard Subject Types

These subject types MUST be used with a Subject Type Vendor ID of 0x005597.

Subject Type	Meaning
0 (TNC_SUBJECT_UNKNOWN)	For some reason, it is not certain exactly what the Identifier Value is identifying. This situation is fairly common today as many authentication methods do not specify exactly what is being authenticated.
1 (TNC_SUBJECT_MACHINE)	Identifier Value is an identifier for the AR machine
2 (TNC_SUBJECT_USER)	Identifier Value is an identifier for a user of the AR

3.6.11.18.4TCG Standard Authentication Methods

These authentication methods MUST be used with an Authentication Method Vendor ID of 0x005597.

Authentication Method	Meaning
0 (TNC_AUTH_UNKNOWN)	Authentication method is unknown
1 (TNC_AUTH_X509_CERT)	Authentication based on an X.509 certificate
2 (TNC_AUTH_PASSWORD)	Authentication based on a password
3 (TNC_AUTH_SIM)	Authentication based on a SIM or USIM card

3.6.11.18.5Examples

This example indicates that the AR established the username “tom” for an AR user via password authentication. This is a hexadecimal dump showing the bytes that must appear in the AR Identities attribute. Note that the bytes in this attribute are stored in big-endian format, as described in section 3.6.11.18.

```
00 00 00 01 // Identity Count = 1
00 00 55 97 // Identity Type Vendor ID = TCG
00 00 00 05 // Identity Type = TNC_ID_USERNAME
00 00 00 03 // Identity Value Length = 3 decimal
74 6f 6d // Identity Value = "tom"
00 00 55 97 // Subject Type Vendor ID = TCG
00 00 00 02 // Subject Type = TNC_SUBJECT_USER
00 00 55 97 // Authentication Method Vendor ID = TCG
00 00 00 02 // Authentication Method = TNC_AUTH_PASSWORD
```

This example indicates that the AR established the DNS name “server5.example.com” for the machine via an X.509 certificate.

```
00 00 00 01 // Identity Count = 1
00 00 55 97 // Identity Type Vendor ID = TCG
00 00 00 03 // Identity Type = TNC_ID_FQDN
00 00 00 13 // Identity Value Length = 19 decimal
73 65 72 76 // Identity Value = "server5.example.com"
65 72 35 2e
65 78 61 6d
70 6c 65 2e
```

```
63 6f 6d
00 00 55 97 // Subject Type Vendor ID = TCG
00 00 00 01 // Subject Type = TNC_SUBJECT_MACHINE
00 00 55 97 // Authentication Method Vendor ID = TCG
00 00 00 01 // Authentication Method = TNC_AUTH_X509_CERT
```

This example indicates that the AR established the email address “tom@example.com” for an AR user via an X.509 certificate and the DNS name “pc45.example.com” for the machine via an X.509 certificate. This would generally require two X.509 certificates.

```
00 00 00 02 // Identity Count = 2
00 00 55 97 // Identity Type Vendor ID = TCG
00 00 00 05 // Identity Type = TNC_ID_USERNAME
00 00 00 0f // Identity Value Length = 15 decimal
74 6f 6d 40 // Identity Value =  tom@example.com
65 78 61 6d
70 6c 65 2e
63 6f 6d
00 00 55 97 // Subject Type Vendor ID = TCG
00 00 00 02 // Subject Type = TNC_SUBJECT_USER
00 00 55 97 // Authentication Method Vendor ID = TCG
00 00 00 01 // Authentication Method = TNC_AUTH_X509_CERT
00 00 55 97 // Identity Type Vendor ID = TCG
00 00 00 03 // Identity Type = TNC_ID_FQDN
00 00 00 10 // Identity Value Length = 16 decimal
70 63 34 35 // Identity Value = "pc45.example.com"
2e 65 78 61
6d 70 6c 65
2e 63 6f 6d
00 00 55 97 // Subject Type Vendor ID = TCG
00 00 00 01 // Subject Type = TNC_SUBJECT_MACHINE
00 00 55 97 // Authentication Method Vendor ID = TCG
00 00 00 01 // Authentication Method = TNC_AUTH_X509_CERT
```

This example indicates that the AR did not establish any identities.

```
00 00 00 00 // Identity Count = 0
```

3.7 Mandatory and Optional Functions

Some of the functions in the IF-IMV API are marked as mandatory below. Mandatory functions MUST be implemented. The rest are marked as optional and need not be implemented. An IMV or TNC Server MUST work properly if one or more optional functions are not implemented by the other party. To determine whether an optional function has been implemented, use the Dynamic Function Binding mechanism defined in most platform bindings. On platforms that don’t define a Dynamic Function Binding mechanism, all optional functions MUST be implemented.

3.8 IMV Functions

These functions are implemented by the IMV and called by the TNC Server.

3.8.1 TNC_IMV_Initialize (MANDATORY)

```
TNC_Result TNC_IMV_Initialize(
      /*in*/  TNC_IMVID imvID,
      /*in*/  TNC_Version minVersion,
      /*in*/  TNC_Version maxVersion,
      /*out*/ TNC_Version *pOutActualVersion);
```

Description:

The TNC Server calls this function to initialize the IMV and agree on the API version number to be used. It also supplies the IMV ID, an IMV identifier that the IMV must use when calling TNC Server callback functions. All IMVs MUST implement this function.

The TNC Server MUST NOT call any other IF-IMV API functions for an IMV until it has successfully completed a call to `TNC_IMV_Initialize`(). Once a call to this function has completed successfully, this function MUST NOT be called again for a particular IMV-TNCS pair until a call to `TNC_IMV_Terminate` has completed successfully.

The TNC Server MUST set `minVersion` to the minimum IF-IMV API version number that it supports and MUST set `maxVersion` to the maximum API version number that it supports. The TNC Server also MUST set `pOutActualVersion` so that the IMV can use it as an output parameter to provide the actual API version number to be used. With the C binding, this would involve setting `pOutActualVersion` to point to a suitable storage location.

The IMV MUST check these to determine whether there is an API version number that it supports in this range. If not, the IMV MUST return `TNC_RESULT_NO_COMMON_VERSION`. Otherwise, the IMV SHOULD select a mutually supported version number, store that version number at `pOutActualVersion`, and initialize the IMV. If the initialization completes successfully, the IMV SHOULD return `TNC_RESULT_SUCCESS`. Otherwise, it SHOULD return another result code.

If an IMV determines that `pOutActualVersion` is not set properly to allow the IMV to use it as an output parameter, the IMV SHOULD return `TNC_RESULT_INVALID_PARAMETER`. With the C binding, this might involve checking for a NULL pointer. IMVs are not required to make this check and there is no guarantee that IMVs will be able to perform it adequately (since it is often impossible or very hard to detect invalid pointers).

Input Parameter	Description
`imvID`	IMV ID assigned by TNCS
`minVersion`	Minimum API version supported by TNCS
`maxVersion`	Maximum API version supported by TNCS

Output Parameter	Description
`pOutActualVersion`	Mutually supported API version number

Result Code	Condition
`TNC_RESULT_SUCCESS`	Success
`TNC_RESULT_NO_COMMON_VERSION`	No common API version supported by IMV and TNC Server
`TNC_RESULT_ALREADY_INITIALIZED`	`TNC_IMV_Initialize` has already been called and `TNC_IMV_Terminate` has not
`TNC_RESULT_INVALID_PARAMETER`	Invalid function parameter
`TNC_RESULT_OTHER`	Unspecified non-fatal error
Other result codes	Other non-fatal error

3.8.2 TNC_IMV_NotifyConnectionChange (OPTIONAL)

```
TNC_Result TNC_IMV_NotifyConnectionChange(
      /*in*/  TNC_IMVID imvID,
      /*in*/  TNC_ConnectionID connectionID
      /*in*/  TNC_ConnectionState newState);
```

Description:

The TNC Server calls this function to inform the IMV that the state of the network connection identified by `connectionID` has changed to `newState`. Section 3.6.5 lists all the possible values of `newState` for this version of the IF-IMV API. The TNCS MUST NOT use any other values with this version of IF-IMV.

IMVs that want to track the state of network connections or maintain per-connection data structures SHOULD implement this function. Other IMVs MAY implement it.

If the state is `TNC_CONNECTION_STATE_CREATE`, the IMV SHOULD note the creation of a new network connection.

If the state is `TNC_CONNECTION_STATE_HANDSHAKE`, an Integrity Check Handshake is about to begin.

If the state is `TNC_CONNECTION_STATE_DELETE`, the IMV SHOULD discard any state pertaining to this network connection and MUST NOT pass this network connection ID to the TNC Server after this function returns (unless the TNCS later creates another network connection with the same network connection ID).

In the `imvID` parameter, the TNCS MUST pass the IMV ID value provided to `TNC_IMV_Initialize`. In the `connectionID` parameter, the TNCS MUST pass a valid network connection ID. IMVs MAY check these values to make sure they are valid and return an error if not, but IMVs are not required to make these checks. In the `newState` parameter, the TNCS MUST pass one of the values listed in section 3.6.5.

Input Parameter	Description
`imvID`	IMV ID assigned by TNCS
`connectionID`	Network connection ID whose state is changing
`newState`	New network connection state

Result Code	Condition
`TNC_RESULT_SUCCESS`	Success
`TNC_RESULT_NOT_INITIALIZED`	`TNC_IMV_Initialize` has not been called
`TNC_RESULT_INVALID_PARAMETER`	Invalid function parameter
`TNC_RESULT_OTHER`	Unspecified non-fatal error
`TNC_RESULT_FATAL`	Unspecified fatal error
Other result codes	Other non-fatal error

3.8.3 TNC_IMV_BeginHandshake (OPTIONAL)

```
TNC_Result TNC_IMV_BeginHandshake(
      /*in*/  TNC_IMVID imvID,
      /*in*/  TNC_ConnectionID connectionID);
```

Description:

The TNC Server calls this function to indicate that an Integrity Check Handshake is beginning and to solicit messages from IMVs for the first batch. The IMV SHOULD send any IMC-IMV messages it wants to send, as soon as possible after this function is called, and then return from this function to indicate that it is finished sending messages for this batch.

This is an optional function, so IMVs are not required to implement it. An IMV should implement this function if it supports having the TNCS send the first batch of IMC-IMV messages in the TNC handshake. This is useful when an IMV must send instructions to the IMC before the IMC can send useful integrity messages. Simple IMVs that do not wish to send an initial message to the IMC need not implement this function.

Since this function was not included in IF-IMV 1.0 through 1.2, many IMVs do not implement it. TNCSs MUST work properly if an IMV does not implement this function. Likewise, many TNCSs do not implement this function, so IMVs MUST work properly if a TNCS does not implement this function. The TNCS SHOULD use dynamic function binding (on platforms where that is available) to determine whether the IMV implements this function. If an IMV implements this function, the TNCS SHOULD call this function if the TNCS is preparing to send the first batch of messages in a TNC handshake. Even if the TNCS and IMV both support this function and have used it for previous handshakes, a particular handshake may not involve having the TNCS send the first batch of messages. In that case, this function will not be called for that handshake. The TNCS and IMV MUST properly handle the situation where this function is supported by both the TNCS and IMV but is not called for a particular handshake because the TNCS will not be sending the first batch of messages.

As with all IMV functions, the IMV SHOULD NOT wait a long time (definitely not more than a second) before returning from `TNC_IMV_BeginHandshake`. To do otherwise would risk delaying the handshake indefinitely. A long delay might frustrate users or exceed network timeouts (PDP, PEP or otherwise).

In the `imvID` parameter, the TNCS MUST pass the IMV ID value provided to `TNC_IMV_Initialize`. In the `connectionID` parameter, the TNCS MUST pass a valid network connection ID. IMVs MAY check these values to make sure they are valid and return an error if not, but IMVs are not required to make these checks.

Input Parameter	Description
`imvID`	IMV ID assigned by TNCS
`connectionID`	Network connection ID on which message was received

Result Code	Condition
`TNC_RESULT_SUCCESS`	Success
`TNC_RESULT_NOT_INITIALIZED`	`TNC_IMV_Initialize` has not been called
`TNC_RESULT_INVALID_PARAMETER`	Invalid function parameter
`TNC_RESULT_OTHER`	Unspecified non-fatal error
`TNC_RESULT_FATAL`	Unspecified fatal error
Other result codes	Other non-fatal error

3.8.4 TNC_IMV_ReceiveMessage (OPTIONAL)

```
TNC_Result TNC_IMV_ReceiveMessage(
      /*in*/  TNC_IMVID imvID,
      /*in*/  TNC_ConnectionID connectionID,
      /*in*/  TNC_BufferReference message,
      /*in*/  TNC_UInt32 messageLength,
      /*in*/  TNC_MessageType messageType);
```

Description:

The TNC Server calls this function to deliver a message to the IMV. The message is contained in the buffer referenced by message and contains the number of octets (bytes) indicated by messageLength. The type of the message is indicated by messageType. The message must be from an IMC (or a TNCC or other party acting as an IMC).

If IF-TNCCS-SOH is used for a connection, and the IMV and TNCS implement `TNC_IMV_ReceiveMessageSOH`, the TNCS SHOULD use that function to deliver the contents of SOHReportEntries instead of using `TNC_IMV_ReceiveMessage`. However, if IF-TNCCS-SOH is used for a connection, but either the IMV or the TNCS does not implement `TNC_IMV_ReceiveMessageSOH`, `TNC_IMV_ReceiveMessage` SHOULD be used instead. For each SOHReportEntry, the value contained in the first System-Health-ID attribute should be compared to the `TNC_MessageType` values previously supplied in the IMV's most recent call to `TNC_TNCS_ReportMessageTypes` or `TNC_TNCS_ReportMessageTypesLong`. The SOHReportEntry should be delivered to each IMV that has a match. If an IMV does not support `TNC_IMV_ReceiveMessageSOH` (or when the TNCS does not support that function), `TNC_IMV_ReceiveMessage` SHOULD be employed. In that case, the TNCS MUST pass in the `message` parameter a reference to a buffer containing the Data field of the first Vendor-Specific attribute whose Vendor ID matches the value contained in the System-Health-ID. If no such Vendor-Specific attribute exists, the SOHReportEntry MUST NOT be delivered to this IMV. The TNCS MUST pass in the `messageLength` parameter the number of octets in this buffer. And the TNCS MUST pass in the `messageType` parameter the value contained in the first System-Health-ID attribute.

The IMV SHOULD send any IMC-IMV messages it wants to send as soon as possible after this function is called and then return from this function to indicate that it is finished sending messages in response to this message.

As with all IMV functions, the IMV SHOULD NOT wait a long time (definitely not more than a second) before returning from `TNC_IMV_ReceiveMessage`. To do otherwise would risk delaying the handshake indefinitely. A long delay might frustrate users or exceed network timeouts (PDP, PEP or otherwise).

The IMV should implement this function if it wants to receive messages. Most IMVs will do so, since they will base their IMV Action Recommendations on measurements received from the IMC. However, some IMVs may base their IMV Action Recommendations on other data such as reports from intrusion detection systems or scanners. Those IMVs need not implement this function.

The IMV MUST NOT ever modify the buffer contents and MUST NOT access the buffer after `TNC_IMV_ReceiveMessage` has returned. If the IMV wants to retain the message, it should copy it before returning from `TNC_IMV_ReceiveMessage`.

In the `imvID` parameter, the TNCS MUST pass the IMV ID value provided to `TNC_IMV_Initialize`. In the `connectionID` parameter, the TNCS MUST pass a valid

network connection ID. In the `message` parameter, the TNCS MUST pass a reference to a buffer containing the message being delivered to the IMV. In the `messageLength` parameter, the TNCS MUST pass the number of octets in the message. If the value of the `messageLength` parameter is zero (0), the `message` parameter may be `NULL` with platform bindings that have such a value. In the `messageType` parameter, the TNCS MUST pass the type of the message. This value MUST match one of the `TNC_MessageType` values previously supplied by the IMV to the TNCS in the IMV's most recent call to `TNC_TNCS_ReportMessageTypes` or `TNC_TNCS_ReportMessageTypesLong`. IMVs MAY check these parameters to make sure they are valid and return an error if not, but IMVs are not required to make these checks.

Input Parameter	Description
`imvID`	IMV ID assigned by TNCS
`connectionID`	Network connection ID on which message was received
`Message`	Reference to buffer containing message
`messageLength`	Number of octets in message
`messageType`	Message type of message

Result Code	Condition
`TNC_RESULT_SUCCESS`	Success
`TNC_RESULT_NOT_INITIALIZED`	`TNC_IMV_Initialize` has not been called
`TNC_RESULT_INVALID_PARAMETER`	Invalid function parameter
`TNC_RESULT_OTHER`	Unspecified non-fatal error
`TNC_RESULT_FATAL`	Unspecified fatal error
Other result codes	Other non-fatal error

3.8.5 TNC_IMV_ReceiveMessageSOH (OPTIONAL)

```
TNC_Result TNC_IMV_ReceiveMessageSOH(
      /*in*/  TNC_IMVID imvID,
      /*in*/  TNC_ConnectionID connectionID,
      /*in*/  TNC_BufferReference sohReportEntry,
      /*in*/  TNC_UInt32 sohRELength,
      /*in*/  TNC_MessageType systemHealthID);
```

Description:

The TNC Server calls this function (if supported by the TNCS and implemented by an IMV) to deliver an SoHReportEntry message to the IMV. This allows an IMV to receive the entire SoHReportEntry instead of just the contents of the Vendor-Specific attribute, as would be the case if the `TNC_IMV_ReceiveMessage` function was used.

A TNCS that supports IF-TNCCS-SOH SHOULD support this function, and a TNCS that supports this function SHOULD call this function for a particular IMV and connection instead of calling `TNC_IMV_ReceiveMessage` for delivering the contents of an SoHReportEntry if the IMV implements this function and the connection uses IF-TNCCS-SOH. A TNCS MUST NOT call this

function for a particular IMV and connection if the IMV does not implement this function, or the connection does not use IF-TNCCS-SOH. A TNCS will often find that some IMVs loaded by that TNCS implement this function and some do not. In that case, when the TNCS is handling a connection that uses IF-TNCCS-SOH, the TNCS would call `TNC_IMV_ReceiveMessageSOH` for the IMVs that implement this function and `TNC_IMV_ReceiveMessage` for the IMVs that do not implement `TNC_IMV_ReceiveMessageSOH`. A TNCS MUST NOT call both `TNC_IMV_ReceiveMessage` and `TNC_IMV_ReceiveMessageSOH` for a single SoHReportEntry for a single IMV.

IMVs are not required to implement this function. An IMV should implement `TNC_IMV_ReceiveMessageSOH` if it wants to receive the entire SoHReportEntry instead of just the Vendor-Specific attribute when IF-TNCCS-SOH is used. However, IMVs should recognize that many TNCSs do not support IF-TNCCS-SOH, and some TNCSs that do support IF-TNCCS-SOH will not support this function. Therefore, IMVs should be prepared to receive messages via `TNC_IMV_ReceiveMessage` in some cases when IF-TNCC-SOH is used. Simple IMVs need not implement this function. IMV implementers may find that implementing `TNC_IMV_ReceiveMessageSOH` is more complex than implementing `TNC_IMV_ReceiveMessage`, since `TNC_IMV_ReceiveMessageSOH` must parse the SoHReportEntry, while `TNC_IMV_ReceiveMessage` only needs to parse the contents of the Vendor-Specific attribute within the SoHReportEntry. For many IMVs, the contents of the Vendor-Specific attribute is all that they need. Therefore, IMVs are not required to implement `TNC_IMV_ReceiveMessageSOH`. TNCSs MUST work properly if an IMV does not implement this function.

The contents of the SoHReportEntry are contained in the buffer referenced by `sohReportEntry`, which contains the number of octets (bytes) indicated by `sohRELength`. The content of the first System-Health-ID attribute in the SOHReportEntry is indicated by `systemHealthID`.

The IMV SHOULD send all IMC-IMV messages intended for this connection after this function is called, since IF-TNCCS-SOH supports only one round-trip.

As with all IMV functions, the IMV SHOULD NOT wait a long time (definitely not more than a second) before returning from `TNC_IMV_ReceiveMessageSOH`. To do otherwise would risk delaying the handshake indefinitely. A long delay might frustrate users or exceed network timeouts (PDP, PEP or otherwise).

The IMV MUST NOT ever modify the buffer contents and MUST NOT access the buffer after `TNC_IMV_ReceiveMessageSOH` has returned. If the IMV wants to retain the content of the SoHReportEntry, the IMC should copy it before returning from `TNC_IMV_ReceiveMessageSOH`.

In the `imvID` parameter, the TNCS MUST pass the IMV ID value provided to `TNC_IMV_Initialize`. In the `connectionID` parameter, the TNCS MUST pass a valid network connection ID for a connection that uses IF-TNCCS-SOH. In the `sohReportEntry` parameter, the TNCS MUST pass a reference to a buffer containing the SoHReportEntry being delivered to the IMV. In the `sohRELength` parameter, the TNCS MUST pass the number of octets in the SoHReportEntry. The value of the `sohRELength` parameter MUST NOT be zero (0) since this is not permitted with the IF-TNCCS-SOH protocol. In the `systemHealthID` parameter, the TNCS MUST pass the contents of the first System-Health-ID attribute in the SoHReportEntry. This value MUST match one of the `TNC_MessageType` values previously supplied by the IMV to the TNCS in the IMV's most recent call to `TNC_TNCS_ReportMessageTypes` or `TNC_TNCS_ReportMessageTypesLong`. IMVs MAY check these parameters to make sure they are valid and return an error if not, but IMVs are not required to make these checks.

Input Parameter	Description
`imvID`	IMV ID assigned by TNCS

connectionID	Network connection ID on which message was received
sohReportEntry	Reference to buffer containing SoHReportEntry
sohRELength	Number of octets in SoHReportEntry
systemHealthID	Contents of first System-Health-ID attribute in SoHReportEntry

Result Code	Condition
TNC_RESULT_SUCCESS	Success
TNC_RESULT_NOT_INITIALIZED	TNC_IMV_Initialize has not been called
TNC_RESULT_INVALID_PARAMETER	Invalid function parameter
TNC_RESULT_OTHER	Unspecified non-fatal error
TNC_RESULT_FATAL	Unspecified fatal error
Other result codes	Other non-fatal error

3.8.6 TNC_IMV_ReceiveMessageLong (OPTIONAL)

```
TNC_Result TNC_IMV_ReceiveMessageLong(
      /*in*/  TNC_IMVID imvID,
      /*in*/  TNC_ConnectionID connectionID,
      /*in*/  TNC_UInt32 messageFlags,
      /*in*/  TNC_BufferReference message,
      /*in*/  TNC_UInt32 messageLength,
      /*in*/  TNC_VendorID messageVendorID,
      /*in*/  TNC_MessageSubtype messageSubtype,
      /*in*/  TNC_UInt32 sourceIMCID,
      /*in*/  TNC_UInt32 destinationIMVID);
```

Description:

The TNC Server calls this function to deliver a message to the IMV. This function provides several features that TNC_IMV_ReceiveMessage does not: longer (32 bit) vendor ID and message subtype fields, a messageFlags field, and the ability to specify a source IMC ID and destination IMV ID.

The message is contained in the buffer referenced by message and contains the number of octets (bytes) indicated by messageLength. The type of the message is indicated by the messageVendorID and messageSubtype parameters. The message must be from an IMC (or a TNCC or other party acting as an IMC). Any flags associated with the message are included in the messageFlags parameter. The sourceIMCID and destinationIMVID parameters indicate the IMC ID of the IMC that sent this message (if available) and either the IMV ID of the intended recipient (if the EXCL flag is set), or the IMV ID in response to whose message or messages this message was sent. If the EXCL flag is set, destinationIMVID MUST be either the primary IMV ID provided to the IMV in TNC_IMV_Initialize, or an additional IMV ID reserved when the IMV requested the TNCS to do so by calling TNC_TNCS_ReserveAdditionalIMVID. If the EXCL flag is not set, then destinationIMVID MAY be set to the wild card TNC_IMVID_ANY.

If an IF-TNCCS protocol that supports long types or exclusive delivery is used for a connection, and the IMV and TNCS implement `TNC_IMV_ReceiveMessageLong`, the TNCS SHOULD use this function to deliver messages instead of using `TNC_IMV_ReceiveMessage`. However, if the IMV does not implement `TNC_IMV_ReceiveMessageLong`, the TNCS SHOULD use `TNC_IMV_ReceiveMessage` instead. Messages whose vendor ID or message subtype is too long to be represented in the parameters supported by `TNC_IMV_ReceiveMessage` MUST NOT be delivered to an IMV that does not support `TNC_IMV_ReceiveMessageLong`. This should be fine, since the IMV in question wouldn't have the ability to process such messages. Also, the IMV probably calls `TNC_TNCS_ReportMessageTypes` instead of `TNC_TNCS_ReportMessageTypesLong`, so it couldn't have expressed an interest in messages with long types except via wild cards. Messages with flags or source IMC IDs or destination IMV IDs can be handled using the `TNC_IMV_ReceiveMessage` function.

The IMV SHOULD send any IMC-IMV messages it wants to send, as soon as possible after this function is called, and then return from this function to indicate that it is finished sending messages in response to this message.

As with all IMV functions, the IMV SHOULD NOT wait a long time (definitely not more than a second) before returning from `TNC_IMV_ReceiveMessageLong`. To do otherwise would risk delaying the handshake indefinitely. A long delay might frustrate users or exceed network timeouts (PDP, PEP or otherwise).

This is an optional function, so IMVs are not required to implement it. An IMV should implement this function if it wants to receive messages with long types, or flags, or source IMC IDs and destination IMV IDs. Simple IMVs need not implement this function. Since this function was not included in IF-IMV 1.0 through 1.2, many IMVs do not implement it. TNCSs MUST work properly if an IMV does not implement this function. Likewise, many TNCSs do not implement this function, so IMVs MUST work properly if a TNCS does not implement this function. The TNCS SHOULD use dynamic function binding (on platforms where that is available) to determine whether the IMV implements this function.

The IMV MUST NOT ever modify the buffer contents and MUST NOT access the buffer after `TNC_IMV_ReceiveMessageLong` has returned. If the IMV wants to retain the message, the IMV should copy it before returning from `TNC_IMV_ReceiveMessageLong`.

In the `imvID` parameter, the TNCS MUST pass the IMV ID value provided to `TNC_IMV_Initialize`. In the `connectionID` parameter, the TNCS MUST pass a valid network connection ID. In the `message` parameter, the TNCS MUST pass a reference to a buffer containing the message being delivered to the IMV. In the `messageLength` parameter, the TNCS MUST pass the number of octets in the message. If the value of the `messageLength` parameter is zero (0), the `message` parameter may be `NULL` with platform bindings that have such a value. In the `messageVendorID` and `messageSubtype` parameters, the TNCS MUST pass the vendor ID and message subtype of the message. These values MUST match one of the values previously supplied by the IMV to the TNCS in the IMV's most recent call to `TNC_TNCS_ReportMessageTypes` or `TNC_TNCS_ReportMessageTypesLong`. The TNCS MUST NOT specify a message type whose vendor ID is 0xffffff or whose subtype is 0xff. These values are reserved for use as wild cards, as described in section 3.9.1. IMVs MAY check these parameters to make sure they are valid and return an error if not, but IMVs are not required to make these checks.

Any flags associated with the message are included in the `messageFlags` parameter. This may include the EXCL flag, but the TNCS MUST process that flag itself to ensure that a message with this flag set is only delivered to the intended recipient.

Input Parameter	**Description**
`imvID`	IMV ID assigned by TNCS

connectionID	Network connection ID on which message was received
messageFlags	Flags associated with message
Message	Reference to buffer containing message
messageLength	Number of octets in message
messageVendorID	Vendor ID associated with message
messageSubtype	Message subtype associated with message
sourceIMCID	Source IMC ID for message
destinationIMVID	Destination IMV ID for message

Result Code	Condition
TNC_RESULT_SUCCESS	Success
TNC_RESULT_INVALID_PARAMETER	Invalid function parameter
TNC_RESULT_ILLEGAL_OPERATION	Message send attempted at illegal time
TNC_RESULT_EXCEEDED_MAX_ROUND_TRIPS	Maximum round trips exceeded
TNC_RESULT_EXCEEDED_MAX_MESSAGE_SIZE	Maximum message size exceeded
TNC_RESULT_OTHER	Unspecified non-fatal error
TNC_RESULT_FATAL	Unspecified fatal error
Other result codes	Other non-fatal error

3.8.7 TNC_IMV_SolicitRecommendation (MANDATORY)

```
TNC_Result TNC_IMV_SolicitRecommendation(
      /*in*/  TNC_IMVID imvID,
      /*in*/  TNC_ConnectionID connectionID);
```

Description:

The TNC Server calls this function at the end of an Integrity Check Handshake (after all IMC-IMV messages have been delivered) to solicit recommendations from IMVs that have not yet provided a recommendation. The TNCS SHOULD NOT call this method for an IMV and a particular connection if that IMV has already called TNC_TNCS_ProvideRecommendation with that connection since the TNCS last called TNC_IMV_NotifyConnectionChange for that IMV and connection. If an IMV is not able to provide a recommendation at this time, it SHOULD call TNC_TNCS_ProvideRecommendation with the recommendation parameter set to TNC_IMV_ACTION_RECOMMENDATION_NO_RECOMMENDATION. If an IMV returns from this function without calling TNC_TNCS_ProvideRecommendation, the TNCS MAY consider the IMV's Action Recommendation to be TNC_IMV_ACTION_RECOMMENDATION_NO_RECOMMENDATION. The TNCS MAY take other actions, such as logging this IMV behavior, which is erroneous.

All IMVs MUST implement this function.

Note that a TNCC or TNCS MAY cut off IMC-IMV communications at any time for any reason, including limited support for long conversations in underlying protocols, user or administrator intervention, or policy. If this happens, the TNCS will return `TNC_RESULT_ILLEGAL_OPERATION` from `TNC_TNCS_SendMessage` and call `TNC_IMV_SolicitRecommendation` to elicit IMV Action Recommendations based on the data they have gathered so far.

In the `imvID` parameter, the TNCS MUST pass the IMV ID value provided to `TNC_IMV_Initialize`. In the `connectionID` parameter, the TNCS MUST pass a valid network connection ID. IMVs MAY check these values to make sure they are valid and return an error if not, but IMVs are not required to make these checks.

Input Parameter	Description
`imvID`	IMV ID assigned by TNCS
`connectionID`	Network connection ID for which a recommendation is requested

Result Code	Condition
`TNC_RESULT_SUCCESS`	Success
`TNC_RESULT_NOT_INITIALIZED`	`TNC_IMV_Initialize` has not been called
`TNC_RESULT_INVALID_PARAMETER`	Invalid function parameter
`TNC_RESULT_OTHER`	Unspecified non-fatal error
`TNC_RESULT_FATAL`	Unspecified fatal error
Other result codes	Other non-fatal error

3.8.8 TNC_IMV_BatchEnding (OPTIONAL)

```
TNC_Result TNC_IMV_BatchEnding(
      /*in*/  TNC_IMVID imvID,
      /*in*/  TNC_ConnectionID connectionID);
```

Description:

The TNC Server calls this function to notify IMVs that all IMC messages received in a batch have been delivered and this is the IMV's last chance to send a message in the batch of IMV messages currently being collected. An IMV MAY implement this function if it wants to perform some actions after all the IMC messages received during a batch have been delivered (using `TNC_IMV_ReceiveMessage`, `TNC_IMV_ReceiveMessageSOH`, or `TNC_IMV_ReceiveMessageLong`). For instance, if an IMV has not received any messages from an IMC it may conclude that its IMC is not installed on the endpoint and may decide to call `TNC_TNCS_ProvideRecommendation` with the `recommendation` parameter set to `TNC_IMV_ACTION_RECOMMENDATION_NO_ACCESS`.

An IMV MAY call `TNC_TNCS_SendMessage` from this function. As with all IMV functions, the IMV SHOULD NOT wait a long time (definitely not more than a second) before returning from `TNC_IMV_BatchEnding`. To do otherwise would risk delaying the handshake indefinitely. A long delay might frustrate users or exceed network timeouts (PDP, PEP or otherwise).

In the `imvID` parameter, the TNCS MUST pass the IMV ID value provided to `TNC_IMV_Initialize`. In the `connectionID` parameter, the TNCS MUST pass a valid

network connection ID. IMVs MAY check these values to make sure they are valid and return an error if not, but IMVs are not required to make these checks.

Input Parameter	Description
imvID	IMV ID assigned by TNCS
connectionID	Network connection ID for which a batch is ending

Result Code	Condition
TNC_RESULT_SUCCESS	Success
TNC_RESULT_NOT_INITIALIZED	TNC_IMV_Initialize has not been called
TNC_RESULT_INVALID_PARAMETER	Invalid function parameter
TNC_RESULT_OTHER	Unspecified non-fatal error
TNC_RESULT_FATAL	Unspecified fatal error
Other result codes	Other non-fatal error

3.8.9 TNC_IMV_Terminate (OPTIONAL)

```
TNC_Result TNC_IMV_Terminate(
      /*in*/  TNC_IMVID imvID);
```

Description:

The TNC Server calls this function to close down the IMV. For example, this function will typically be called when all work is complete and the TNCS is preparing to shut down or when the IMV reports TNC_RESULT_FATAL. Once a call to TNC_IMV_Terminate is made, the TNC Server MUST NOT call the IMV except to call TNC_IMV_Initialize (which may not succeed if the IMV cannot reinitialize itself). Even if the IMV returns an error from this function, the TNC Server MAY continue with its unload or shutdown procedure.

In the imvID parameter, the TNCS MUST pass the IMV ID value provided to TNC_IMV_Initialize. IMVs MAY check if imvID matches the value previously passed to TNC_IMV_Initialize and return TNC_RESULT_INVALID_PARAMETER if not, but they are not required to make this check.

Input Parameter	Description
imvID	IMV ID assigned by TNCS

Result Code	Condition
TNC_RESULT_SUCCESS	Success
TNC_RESULT_NOT_INITIALIZED	TNC_IMV_Initialize has not been called
TNC_RESULT_INVALID_PARAMETER	Invalid function parameter
TNC_RESULT_OTHER	Unspecified non-fatal error
TNC_RESULT_FATAL	Unspecified fatal error

Other result codes	Other non-fatal error

3.9 TNC Server Functions

These functions are implemented by the TNC Server and called by the IMV.

3.9.1 TNC_TNCS_ReportMessageTypes (MANDATORY)

```
TNC_Result TNC_TNCS_ReportMessageTypes(
      /*in*/  TNC_IMVID imvID,
      /*in*/  TNC_MessageTypeList supportedTypes,
      /*in*/  TNC_UInt32 typeCount);
```

Description:

An IMV calls this function to inform a TNCS about the set of message types the IMV is able to receive. Often, the IMV will call this function from `TNC_IMV_Initialize`. With the Windows DLL binding or UNIX/Linux Dynamic Linkage binding, `TNC_TNCS_ReportMessageTypes` will typically be called from `TNC_IMV_ProvideBindFunction` since an IMV cannot call the TNCS with those platform bindings until `TNC_IMV_ProvideBindFunction` is called.

A list of message types is contained in the `supportedTypes` parameter. The number of types in the list is contained in the `typeCount` parameter. If the value of the `typeCount` parameter is zero (0), the `supportedTypes` parameter may be `NULL` with platform bindings that have such a value. In the `imvID` parameter, the IMV MUST pass the value provided to `TNC_IMV_Initialize`. TNCSs MAY check if `imvID` matches the value previously passed to `TNC_IMV_Initialize` and return `TNC_RESULT_INVALID_PARAMETER` if not, but they are not required to make this check.

All TNC Servers MUST implement this function. The TNC Server MUST NOT ever modify the list of message types and MUST NOT access this list after `TNC_TNCS_ReportMessageTypes` has returned. Generally, the TNC Server will copy the contents of this list before returning from this function. TNC Servers MUST support any message type.

Note that although all TNC Servers must implement this function, some IMVs may never call it if they don't support receiving any message types. This is acceptable. In such a case, the TNC Server MUST NOT deliver any messages to the IMV.

If an IMV requests a message type whose vendor ID is `TNC_VENDORID_ANY` and whose subtype is `TNC_SUBTYPE_ANY` it will receive all messages with any message type, except for messages marked for exclusive delivery to another IMV. This message type is `0xffffffff`. If an IMV requests a message type whose vendor ID is NOT `TNC_VENDORID_ANY` and whose subtype is `TNC_SUBTYPE_ANY`, it will receive all messages with the specified vendor ID and any subtype, except for messages marked for exclusive delivery to another IMV. If an IMV calls `TNC_TNCS_ReportMessageTypes` or `TNC_TNCS_ReportMessageTypesLong` more than once, the message type list supplied in the latest call supplants the message type lists supplied in earlier calls.

Input Parameter	Description
`imvID`	IMV ID assigned by TNCS
`supportedTypes`	Reference to list of message types supported by IMV

typeCount	Number of message types supported by IMV

Result Code	Condition
TNC_RESULT_SUCCESS	Success
TNC_RESULT_INVALID_PARAMETER	Invalid function parameter
TNC_RESULT_OTHER	Unspecified non-fatal error
TNC_RESULT_FATAL	Unspecified fatal error
Other result codes	Other non-fatal error

3.9.2 TNC_TNCS_ReportMessageTypesLong (OPTIONAL)

```
TNC_Result TNC_TNCS_ReportMessageTypesLong(
      /*in*/  TNC_IMVID imvID,
      /*in*/  TNC_VendorIDList supportedVendorIDs,
      /*in*/  TNC_MessageSubtypeList supportedSubtypes,
      /*in*/  TNC_UInt32 typeCount);
```

Description:

An IMV calls this function to inform a TNCS about the set of message types the IMV is able to receive. This function supports long message types, unlike TNC_TNCS_ReportMessageTypes.

Often, the IMV will call this function from TNC_IMV_Initialize. With the Windows DLL binding or UNIX/Linux Dynamic Linkage binding, TNC_TNCS_ReportMessageTypesLong will often be called from TNC_IMV_ProvideBindFunction, since an IMV cannot call the TNCS with those platform bindings until TNC_IMV_ProvideBindFunction is called.

A list of Vendor IDs is contained in the supportedVendorIDs parameter and a list of Message Subtypes is contained in the supportedSubtypes parameter. Each of these lists MUST contain exactly the number of entries given in the typeCount parameter. If the value of the typeCount parameter is zero (0), the supportedVendorIDs and supportedSubtypes parameters may be NULL with platform bindings that have such a value. The values in the supportedVendorIDs list and the supportedSubtypes list are matched pairs that represent the (Vendor ID, Message Subtype) pairs that the IMV is able to receive.

In the imvID parameter, the IMV MUST pass the value provided to TNC_IMV_Initialize. TNCSs MAY check if imvID matches the value previously passed to TNC_IMV_Initialize and return TNC_RESULT_INVALID_PARAMETER if not, but they are not required to make this check.

This function is optional and is not supported by all TNCSs. However, a TNCS that indicates that one of its connections supports long types by returning a value of 1 for the Has Long Types attribute MUST support this function. IMVs should recognize that many TNCSs do not support long types and therefore will not support this function. In those cases, IMVs should be prepared to use the TNC_TNCS_ReportMessageTypes function. Simple IMVs that do not send or receive messages with long types need not call this function.

The TNC Server MUST NOT ever modify the lists of vendor IDs and subtypes and MUST NOT access these lists after TNC_TNCS_ReportMessageTypesLong has returned. Generally, the TNC Server will copy the contents of these lists before returning from this function. TNC Servers MUST support any message type, in other words any combination of vendor IDs and subtypes.

If an IMV requests a message type whose vendor ID is TNC_VENDORID_ANY and whose subtype is TNC_SUBTYPE_ANY, it will receive all messages with any message type, except for messages marked for exclusive delivery to another IMV. If an IMV requests a message type whose vendor ID is NOT TNC_VENDORID_ANY and whose subtype is TNC_SUBTYPE_ANY, it will receive all messages with the specified vendor ID and any subtype, except for messages marked for exclusive delivery to another IMV. Each time a particular IMV calls TNC_TNCS_ReportMessageTypes or TNC_TNCS_ReportMessageTypesLong, the message type list supplied in the latest call supplants the message type lists supplied by that IMV in earlier calls.

Input Parameter	Description
imvID	IMV ID assigned by TNCS
supportedVendorIDs	Reference to list of Vendor IDs supported by IMV
supportedSubtypes	Reference to list of Message Subtypes supported by IMV
typeCount	Number of message types supported by IMV

Result Code	Condition
TNC_RESULT_SUCCESS	Success
TNC_RESULT_INVALID_PARAMETER	Invalid function parameter
TNC_RESULT_OTHER	Unspecified non-fatal error
TNC_RESULT_FATAL	Unspecified fatal error
Other result codes	Other non-fatal error

3.9.3 TNC_TNCS_SendMessage (MANDATORY)

```
TNC_Result TNC_TNCS_SendMessage(
      /*in*/  TNC_IMVID imvID,
      /*in*/  TNC_ConnectionID connectionID,
      /*in*/  TNC_BufferReference message,
      /*in*/  TNC_UInt32 messageLength,
      /*in*/  TNC_MessageType messageType);
```

Description:

An IMV calls this function to give a message to the TNCS for delivery. The message is contained in the buffer referenced by the message parameter and contains the number of octets (bytes) indicated by the messageLength parameter. If the value of the messageLength parameter is zero (0), the message parameter may be NULL with platform bindings that have such a value. The type of the message is indicated by the messageType parameter. In the imvID parameter, the IMV MUST pass the value provided to TNC_IMV_Initialize. In the connectionID parameter, the IMV MUST pass a valid network connection ID. TNCSs MAY check these values to make sure they are valid and return an error if not, but TNCSs are not required to make these checks.

All TNC Servers MUST implement this function. The TNC Server MUST NOT ever modify the buffer contents and MUST NOT access the buffer after TNC_TNCS_SendMessage has returned.

The TNC Server will typically copy the message out of the buffer, queue it up for delivery, and return from this function.

The IMV MUST NOT call this function unless it has received a call to TNC_IMV_BeginHandshake, TNC_IMV_ReceiveMessage, TNC_IMV_ReceiveMessageSOH, TNC_IMV_ReceiveMessageLong, or TNC_IMV_BatchEnding for this connection and the IMV has not yet returned from that function. If the IMV violates this prohibition, the TNCS SHOULD return TNC_RESULT_ILLEGAL_OPERATION. If an IMV really wants to communicate with an IMC at another time, it should call TNC_TNCS_RequestHandshakeRetry.

Note that a TNCC or TNCS MAY cut off IMC-IMV communications at any time for any reason, including limited support for long conversations in underlying protocols, user or administrator intervention, or policy. If this happens, the TNCS will return TNC_RESULT_ILLEGAL_OPERATION from TNC_TNCS_SendMessage and call TNC_IMV_SolicitRecommendation to elicit IMV Action Recommendations based on the data they have gathered so far.

If the TNCS supports limiting the message size or number of round trips, and the limits are exceeded, the TNCS MUST return TNC_RESULT_EXCEEDED_MAX_MESSAGE_SIZE or TNC_RESULT_EXCEEDED_MAX_ROUND_TRIPS as the case may be. An IMV can get the Maximum Message Size and Maximum Number of Round Trips by using the related Attribute ID within the TNC_TNCS_GetAttribute function. The IMV SHOULD adapt its behavior to accommodate these limitations if available.

The TNC Server MUST support any message type. However, the IMV MUST NOT specify a message type whose vendor ID is 0xffffff or whose subtype is 0xff. These values are reserved for use as wild cards, as described in section 3.9.1. If the IMV violates this prohibition, the TNCS SHOULD return TNC_RESULT_INVALID_PARAMETER.

Input Parameter	Description
imvID	IMV ID assigned by TNCS
connectionID	Network connection ID on which message should be sent
message	Reference to buffer containing message
messageLength	Number of octets in message
messageType	Message type of message

Result Code	Condition
TNC_RESULT_SUCCESS	Success
TNC_RESULT_INVALID_PARAMETER	Invalid function parameter
TNC_RESULT_ILLEGAL_OPERATION	Message send attempted at illegal time
TNC_RESULT_EXCEEDED_MAX_ROUND_TRIPS	Maximum round trips exceeded
TNC_RESULT_EXCEEDED_MAX_MESSAGE_SIZE	Maximum message size exceeded
TNC_RESULT_OTHER	Unspecified non-fatal error
TNC_RESULT_FATAL	Unspecified fatal error
Other result codes	Other non-fatal error

3.9.4 TNC_TNCS_SendMessageSOH (OPTIONAL)

```
TNC_Result TNC_TNCS_SendMessageSOH(
      /*in*/  TNC_IMVID imvID,
      /*in*/  TNC_ConnectionID connectionID,
      /*in*/  TNC_BufferReference sohrReportEntry,
      /*in*/  TNC_UInt32 sohrRELength);
```

Description:

An IMV calls this function (if implemented by the TNCS) to give an SoHRReportEntry to the TNCS for delivery. This allows an IMV to send an entire SoHRReportEntry instead of just the contents of the Vendor-Specific attribute, as would be the case if the `TNC_TNCS_SendMessage` function was used.

The SoHRReportEntry (as defined in section 3.5 of [12]) is contained in the buffer referenced by the `sohrReportEntry` parameter and contains the number of octets (bytes) indicated by the `sohrRELength` parameter. If the value of the `sohrRELength` parameter is zero (0), the TNCS MUST return `TNC_RESULT_INVALID_PARAMETER`, since a zero-length SoHRReportEntry is prohibited by the IF_TNCCS-SOH 1.0 specification. No type is included with the SoHRReportEntry, since this is included in the System-Health-ID attribute in the SoHRReportEntry. In the `imvID` parameter, the IMV MUST pass the value provided to `TNC_IMV_Initialize`. In the `connectionID` parameter, the IMV MUST pass a valid network connection ID. TNCSs MAY check these values to make sure they are valid and return `TNC_RESULT_INVALID_PARAMETER` if not, but TNCSs are not required to make these checks.

A TNCS that supports IF-TNCCS-SOH SHOULD support this function. The TNC Server MUST NOT ever modify the buffer contents and MUST NOT access the buffer after `TNC_TNCS_SendMessageSOH` has returned. The TNC Server will typically copy the SoHRReportEntry out of the buffer, place it into the SoHR message or queue it up for such placement, and return from this function.

An IMV may call this function if it needs to send an SoHRReportEntry with a connection that supports IF-TNCCS-SOH (as indicated by the Has SOH attribute being set to a value of 1). If an IMV calls this function for a connection that does not support IF-TNCCS-SOH, the TNCS SHOULD return `TNC_RESULT_NO_SOH_SUPPORT` from `TNC_TNCS_SendMessageSOH`. IMVs should recognize that many TNCSs do not support IF-TNCCS-SOH, and some TNCSs that do support IF-TNCCS-SOH will not support this function. Therefore, IMVs should be prepared to send messages via `TNC_TNCS_SendMessage` in some cases when IF-TNCC-SOH is used. Simple IMVs that only need to send simple messages need not call this function. They can just call `TNC_TNCS_SendMessage`, which will automatically place their message in a Vendor-Specific attribute in an SoHRReportEntry.

The IMV MUST NOT call this function unless it has received a call to `TNC_IMV_BeginHandshake`, `TNC_IMV_ReceiveMessage`, `TNC_IMV_ReceiveMessageSOH`, `TNC_IMV_ReceiveMessageLong`, or `TNC_IMV_BatchEnding` for this connection and the IMV has not yet returned from that function. If the IMV violates this prohibition, the TNCS SHOULD return `TNC_RESULT_ILLEGAL_OPERATION`. If an IMV really wants to communicate with an IMC at another time, it should call `TNC_TNCS_RequestHandshakeRetry`.

Note that a TNCC or TNCS MAY cut off IMC-IMV communications at any time for any reason, including limited support for long conversations in underlying protocols, user or administrator intervention, or policy. If this happens, the TNCS will return `TNC_RESULT_ILLEGAL_OPERATION` from `TNC_TNCS_SendMessageSOH` and may call `TNC_IMV_SolicitRecommendation` to elicit IMV Action Recommendations based on the data they have gathered so far.

If the TNCS supports limiting the message size or number of round trips, and the limits are exceeded, the TNCS MUST return TNC_RESULT_EXCEEDED_MAX_MESSAGE_SIZE or TNC_RESULT_EXCEEDED_MAX_ROUND_TRIPS as the case may be. An IMV can get the Maximum Message Size and Maximum Number of Round Trips by using the related Attribute ID within the TNC_TNCS_GetAttribute function. The IMV SHOULD adapt its behavior to accommodate these limitations if available.

A TNCS MAY validate sohrReportEntry in accordance with the format defined in section 3.6.1.3 of [12] but is not required to do so. If sohrReportEntry is not found compliant with the required format, the TNCS MUST return TNC_RESULT_INVALID_PARAMETER.

Input Parameter	Description
imvID	IMV ID assigned by TNCS
connectionID	Network connection ID on which SoHRReportEntry should be sent
sohrReportEntry	Reference to buffer containing SoHRReportEntry
sohrRELength	Number of octets in SoHRReportEntry

Result Code	Condition
TNC_RESULT_SUCCESS	Success
TNC_RESULT_INVALID_PARAMETER	Invalid function parameter
TNC_RESULT_ILLEGAL_OPERATION	Message send attempted at illegal time
TNC_RESULT_EXCEEDED_MAX_ROUND_TRIPS	Maximum round trips exceeded
TNC_RESULT_EXCEEDED_MAX_MESSAGE_SIZE	Maximum message size exceeded
TNC_RESULT_NO_SOH_SUPPORT	Connection does not support SOH
TNC_RESULT_OTHER	Unspecified non-fatal error
TNC_RESULT_FATAL	Unspecified fatal error
Other result codes	Other non-fatal error

3.9.5 TNC_TNCS_SendMessageLong (OPTIONAL)

```
TNC_Result TNC_TNCS_SendMessageLong(
      /*in*/  TNC_IMVID imvID,
      /*in*/  TNC_ConnectionID connectionID,
      /*in*/  TNC_UInt32 messageFlags,
      /*in*/  TNC_BufferReference message,
      /*in*/  TNC_UInt32 messageLength,
      /*in*/  TNC_VendorID messageVendorID,
      /*in*/  TNC_MessageSubtype messageSubtype,
      /*in*/  TNC_UInt32 destinationIMCID);
```

Description:

An IMV calls this function to give a message to the TNCS for delivery. This function provides several features that TNC_TNCS_SendMessage does not: longer 32 bit vendor ID and message subtype fields, a messageFlags field, and the ability to specify a desired or exclusive destination IMC ID.

The message is contained in the buffer referenced by the message parameter and contains the number of octets (bytes) indicated by the messageLength parameter. If the value of the messageLength parameter is zero (0), the message parameter may be NULL with platform bindings that have such a value. The type of the message is indicated by the messageVendorID and messageSubtype parameters. In the imvID parameter, the IMV MUST pass either the primary IMV ID provided to the IMV in TNC_IMV_Initialize, or an additional IMV ID reserved when the IMV requests the TNCS to do so by calling TNC_TNCS_ReserveAdditionalIMVID. The IMV ID passed will be used as the sourceIMVID for protocols that support sending this field along with the message. In the connectionID parameter, the IMV MUST pass a valid network connection ID. TNCSs MAY check these values to make sure they are valid and return an error if not, but TNCSs are not required to make these checks.

The messageFlags parameter supports up to 32 flags that may be set pertaining to the message. At this time, only one flag has been defined. The most significant bit in the messageFlags parameter (i.e. the value 0x80000000) is known as the TNC_MESSAGE_FLAGS_EXCLUSIVE flag ("the EXCL flag", for short). If this bit is set, the sending IMV is requesting that the message only be delivered to the IMC designated by the destinationIMCID parameter. If the bit is cleared, the message is to be delivered to any IMC that has indicated an interest in the message type. Other flags may be defined in future versions of this specification. Until that time, IMVs MUST leave all these flags cleared. Some connections and/or TNCSs may not support the EXCL flag. To determine whether a particular TNCS and connection supports this flag, the IMV should get the Has Exclusive attribute for that connection. If the IMV sets a flag that is not supported by the TNCS or the connection, the TNCS SHOULD return TNC_RESULT_INVALID_PARAMETER.

If the EXCL flag is set, the IMV MUST set the destinationIMCID parameter to indicate which IMC should receive the message being sent. The IMV should place into this parameter a value which it has obtained using the sourceIMCID parameter for messages previously delivered to the IMV on this connection. This ensures that the IMV ID is valid for this connection. The IMV MAY set the destinationIMCID parameter to a specific IMC ID even when the EXCL flag is not set. This indicates that the message being sent is in response to a message received from the IMC indicated in the destinationIMCID parameter. If the IMV does not know the IMC ID of the recipient IMC, the IMV MUST use the TNC_IMCID_ANY wild card for the destinationIMCID parameter and MUST NOT set the EXCL flag. In such a case, the message is delivered to all IMCs that have expressed interest in receiving such messages. This may happen when the TNCS starts the integrity handshake, and the IMV does not know the IMC IDs of the recipient IMCs. IMVs that receive a message with a destinationIMVID value but no EXCL flag set may use the IMV ID to sort out which messages are intended for which IMVs. However, the message type is generally a better way to solve this problem. In fact, there usually is no need for either the EXCL flag or the destinationIMVID parameter. As long as only one IMC has subscribed to each message type, there is no confusion about which message should go to which IMC. Confusion can arise when multiple IMCs are subscribed to the same message type. Since message types are becoming standardized (via IF-M), this is more likely, and therefore the EXCL flag and destinationIMVID parameter are being provided to help sort things out. A TNCS that supports IF-TNCCS 2.0 SHOULD try to assign the same primary IMV ID to an IMV even across restarts, since some TNCSs may use a single connection ID for multiple protocol sessions that cross a TNCS reboot, and an IMC might not realize that IMV IDs have been reassigned. A similar recommendation is contained in IF-IMC regarding the maintenance of IMC ID assignments across TNCC restarts.

This is an optional function, so the TNCS is not required to implement it. Since this function was not included in IF-IMV 1.0 through 1.2, many TNCSs do not implement it. IMVs MUST work properly if a TNCS does not implement this function. The IMV is never required to call this function. The TNCS MUST work with IMVs that don't call this function. The IMV SHOULD use dynamic function binding (on platforms where that is available) to determine whether the TNCS implements this function. The IMV also SHOULD check the Has Long attribute for a given connection to determine whether this connection supports long types before calling this function for that connection. If the connection does not support long types, the IMV MAY call this function (if implemented) but MUST NOT use a `messageVendorID` greater than `0xffffff` or a `messageSubtype` greater than `0xff`. If the IMV does so, the TNCS SHOULD return `TNC_RESULT_NO_LONG_MESSAGE_TYPES`.

The TNC Server MUST NOT ever modify the buffer contents and MUST NOT access the buffer after `TNC_TNCS_SendMessageLong` has returned. The TNC Server will typically copy the message out of the buffer, queue it up for delivery, and return from this function.

The IMV MUST NOT call this function unless it has received a call to `TNC_IMV_BeginHandshake`, `TNC_IMV_ReceiveMessage`, `TNC_IMV_ReceiveMessageSOH`, `TNC_IMV_ReceiveMessageLong`, or `TNC_IMV_BatchEnding` for this connection and has not yet returned from that function. If the IMV violates this prohibition, the TNCS SHOULD return `TNC_RESULT_ILLEGAL_OPERATION`. If an IMV really wants to communicate with an IMC at another time, it should call `TNC_TNCS_RequestHandshakeRetry`.

Note that a TNCC or TNCS MAY cut off IMC-IMV communications at any time for any reason, including limited support for long conversations in underlying protocols, user or administrator intervention, or policy. If this happens, the TNCS will return `TNC_RESULT_ILLEGAL_OPERATION` from `TNC_TNCS_SendMessageLong`. If the TNCS supports limiting the message size or number of round trips, the TNCS MUST return `TNC_RESULT_EXCEEDED_MAX_MESSAGE_SIZE` or `TNC_RESULT_EXCEEDED_MAX_ROUND_TRIPS` respectively if the limits are exceeded. An IMV can get the Maximum Message Size and Maximum Number of Round Trips by using the related Attribute ID within the `TNC_TNCS_GetAttribute` function. The IMV SHOULD adapt its behavior to accommodate these limitations if available.

The TNC Server MUST support any message type. However, the IMV MUST NOT specify a message type whose vendor ID is 0xffffff or whose subtype is 0xff. These values are reserved for use as wild cards, as described in section 3.9.1. If the IMV violates this prohibition, the TNCS SHOULD return `TNC_RESULT_INVALID_PARAMETER`.

Input Parameter	**Description**
`imvID`	IMV ID assigned by TNCS
`connectionID`	Network connection ID on which message should be sent
`messageFlags`	Flags associated with message
`Message`	Reference to buffer containing message
`messageLength`	Number of octets in message
`messageVendorID`	Vendor ID associated with message
`messageSubtype`	Message subtype associated with message
`destinationIMCID`	Destination IMC ID for message

Result Code	**Condition**

TNC_RESULT_SUCCESS	Success
TNC_RESULT_INVALID_PARAMETER	Invalid function parameter
TNC_RESULT_ILLEGAL_OPERATION	Message send attempted at illegal time
TNC_RESULT_EXCEEDED_MAX_ROUND_TRIPS	Maximum round trips exceeded
TNC_RESULT_EXCEEDED_MAX_MESSAGE_SIZE	Maximum message size exceeded
TNC_RESULT_NO_LONG_MESSAGE_TYPES	Long message type used on connection that doesn't support long message types
TNC_RESULT_OTHER	Unspecified non-fatal error
TNC_RESULT_FATAL	Unspecified fatal error
Other result codes	Other non-fatal error

3.9.6 TNC_TNCS_RequestHandshakeRetry (MANDATORY)

```
TNC_Result TNC_TNCS_RequestHandshakeRetry(
      /*in*/  TNC_IMVID imvID,
      /*in*/  TNC_ConnectionID connectionID,
      /*in*/  TNC_RetryReason reason);
```

Description:

An IMV calls this function to ask a TNCS to retry an Integrity Check Handshake. The IMV MUST pass its IMV ID as the `imvID` parameter, a network connection ID as the `connectionID` parameter, and one of the handshake retry reasons listed in section 3.6.6 as the `reason` parameter. If the network connection ID is `TNC_CONNECTIONID_ANY`, then the IMV requests an Integrity Check Handshake retry on all current network connections.

TNCSs MAY check the parameters to make sure they are valid and return an error if not, but TNCSs are not required to make these checks. The `reason` parameter explains why the IMV is requesting a handshake retry. The TNCS MAY use this in deciding whether to attempt the handshake retry. As noted in section 2.9.3, TNCSs are not required to honor IMV requests for handshake retry (especially since handshake retry may not be possible or may interrupt network connectivity). An IMV MAY call this function at any time, even if an Integrity Check Handshake is currently underway. This is useful if the IMV suddenly gets important information but has already finished its dialog with the IMC, for instance. As always, the TNCS is not required to honor the request for handshake retry.

If the TNCS cannot attempt the handshake retry, it SHOULD return the result code `TNC_RESULT_CANT_RETRY`. If the TNCS could attempt to retry the handshake but chooses not to, it SHOULD return the result code `TNC_RESULT_WONT_RETRY`. If the TNCS intends to retry the handshake, it SHOULD return the result code `TNC_RESULT_SUCCESS`. The IMV MAY use this information in displaying diagnostic and progress messages.

If the TNCS supports limiting the message size or number of round trips, the TNCS MUST return TNC_RESULT_EXCEEDED_MAX_MESSAGE_SIZE or TNC_RESULT_EXCEEDED_MAX_ROUND_TRIPS respectively if the limits are exceeded. An IMV can get the Maximum Message Size and Maximum Number of Round Trips by using the related Attribute ID within the TNC_TNCS_GetAttribute function. The IMV SHOULD adapt its behavior to accommodate these limitations if available.

Input Parameter	Description
imvID	IMV ID assigned by TNCS
connectionID	Network connection ID for which handshake retry is requested
reason	Reason why handshake retry is requested

Result Code	Condition
TNC_RESULT_SUCCESS	TNCS intends to retry the handshake
TNC_RESULT_CANT_RETRY	TNCS cannot attempt the handshake retry
TNC_RESULT_WONT_RETRY	TNCS won't attempt the handshake retry
TNC_RESULT_INVALID_PARAMETER	Invalid function parameter
TNC_RESULT_OTHER	Unspecified non-fatal error
TNC_RESULT_FATAL	Unspecified fatal error
Other result codes	Other non-fatal error

3.9.7 TNC_TNCS_ProvideRecommendation (MANDATORY)

```
TNC_Result TNC_TNCS_ProvideRecommendation(
      /*in*/  TNC_IMVID imvID,
      /*in*/  TNC_ConnectionID connectionID,
      /*in*/  TNC_IMV_Action_Recommendation recommendation,
      /*in*/  TNC_IMV_Evaluation_Result evaluation);
```

Description:

An IMV calls this function to deliver its IMV Action Recommendation and IMV Evaluation Result to the TNCS. The TNCS SHOULD use the `recommendation` value in determining its own TNCS Action Recommendation or decision about endpoint access. The TNC specifications do not specify how the TNCS does the `recommendation` value but it is certainly essential to have a recommendation from the IMV. The TNC specifications also do not specify what the TNCS does with the `evaluation` value. It may log it.

The IMV MUST pass its IMV ID as the `imvID` parameter, a valid network connection ID as the `connectionID` parameter, one of the IMV Action Recommendation values listed in section 3.6.7 as the `recommendation` parameter, and one of the IMV Evaluation Result values listed in section 3.6.8 as the `evaluation` parameter. TNCSs MAY check these values to make sure they are valid and return an error if not, but TNCSs are not required to make these checks.

The IMV should deliver its IMV Action Recommendation as soon as possible so that the TNCS can proceed with determining its own TNCS Action Recommendation. If the IMV receives a message from an IMC and is able to decide on an IMV Action Recommendation and deliver it to the TNCS before returning from `TNC_IMV_ReceiveMessage`, it SHOULD do so. However, as always the IMV SHOULD return promptly to avoid a long delay that might frustrate users or exceed network timeouts (PDP, PEP or otherwise).

An IMV SHOULD NOT expect that it will be able to send IMC-IMV messages after calling `TNC_TNCS_ProvideRecommendation`. The TNCS may decide to terminate the handshake

immediately based on the IMV Action Recommendation. For instance, IMVs SHOULD send remediation instructions before calling TNC_TNCS_ProvideRecommendation.

However, a TNCS MAY continue to deliver messages after an IMV calls TNC_TNCS_ProvideRecommendation, especially if other IMVs continue the dialog after the one IMV has rendered its decision. The IMV MUST be prepared for this. It MAY simply ignore these late messages or it MAY consider them and even change its recommendation by calling TNC_TNCS_ProvideRecommendation again. In this case, the TNCS SHOULD use the last recommendation received from an IMV during a particular handshake. However, the TNCS is not required to do this.

If an IMV does not provide a recommendation earlier, the TNCS will call TNC_IMV_SolicitRecommendation at the end of an Integrity Check Handshake (after all IMC-IMV messages have been delivered). The IMV SHOULD then call TNC_TNCS_ProvideRecommendation to deliver its recommendation. If the IMV calls this function when there is no active handshake on the specified network connection, the TNCS SHOULD return TNC_RESULT_ILLEGAL_OPERATION. If an IMV really needs to communicate a recommendation at another time, it should call TNC_TNCS_RequestHandshakeRetry.

Input Parameter	Description
imvID	IMV ID assigned by TNCS
connectionID	Network connection ID for which recommendation is being supplied
recommendation	IMV's Action Recommendation
evaluation	IMV Evaluation Result

Result Code	Condition
TNC_RESULT_SUCCESS	TNCS intends to retry the handshake
TNC_RESULT_INVALID_PARAMETER	Invalid function parameter
TNC_RESULT_ILLEGAL_OPERATION	Recommendation provided at an illegal time
TNC_RESULT_OTHER	Unspecified non-fatal error
TNC_RESULT_FATAL	Unspecified fatal error
Other result codes	Other non-fatal error

3.9.8 TNC_TNCS_GetAttribute (OPTIONAL)

```
TNC_Result TNC_TNCS_GetAttribute(
      /*in*/  TNC_IMVID imvID,
      /*in*/  TNC_ConnectionID connectionID,
      /*in*/  TNC_AttributeID attributeID,
      /*in*/  TNC_UInt32 bufferLength,
      /*out*/ TNC_BufferReference buffer,
      /*out*/ TNC_UInt32 *pOutValueLength);
```

Description:

An IMV calls this function to get the value of an attribute associated with a connection or with the TNCS as a whole. This function is optional. The TNCS is not required to implement it. Since this

function was not included in IF-IMV 1.0, many TNCSs do not support it. IMVs MUST work properly if a TNCS does not implement this function.

The IMV MUST pass its IMV ID as the `imvID` parameter, a standard or vendor-specific attribute ID as the `attributeID` parameter, and a valid network connection ID as the `connectionID` parameter. If the IMV passes a valid connection ID, the TNCS SHOULD provide the attribute value for the specified connection (if supported). If the IMV passes `TNC_CONNECTIONID_ANY`, the TNCS SHOULD provide the value for the TNCS (if supported). If the TNCS does not recognize the attribute ID or connection ID, it SHOULD return the `TNC_RESULT_INVALID_PARAMETER` result code. If the TNCS recognizes the attribute ID and connection ID but does not have an attribute value for the requested attribute ID and connection ID, it SHOULD also return `TNC_RESULT_INVALID_PARAMETER`.

The IMV MUST set `pOutValueLength` so that the TNCS can use it as an output parameter to provide the length in bytes of the requested attribute value. With the C binding, this would involve setting `pOutValueLength` to point to a suitable storage location.

If the TNCS returns a result code other than `TNC_RESULT_SUCCESS`, it MUST NOT store any values via the supplied parameters. But if it returns `TNC_RESULT_SUCCESS`, it MUST provide the length in bytes of the requested attribute value. This length is stored in the manner indicated by the `pOutValueLength` parameter (through a pointer, for the C binding).

If the IMV passes 0 for the `bufferLength` parameter, the TNCS ignores the value of the `buffer` parameter. If the IMV passes a non-zero value for the `bufferLength` parameter, the IMV MUST set the `buffer` parameter so that the TNCS can use it as an output parameter to provide the requested attribute value. The IMV MUST provide enough storage for at least `bufferLength` bytes to be stored via the `buffer` parameter.

The TNCS MUST check the length of the requested attribute value before storing anything via the `buffer` parameter. If the length of the requested attribute value is greater than the `bufferLength` parameter, the TNCS MUST NOT store any data via the `buffer` parameter. Instead, it MUST simply store the length via the `pOutValueLength` parameter. This allows the IMV to recognize that more storage space is needed. In either case, a result code of `TNC_RESULT_SUCCESS` SHOULD be returned.

The TNCS MUST NOT modify the values stored in the `buffer` and `pOutValueLength` parameters after returning from this function. It absolutely MUST NOT store more than `bufferLength` bytes via the `buffer` parameter.

Input Parameter	**Description**
`imvID`	IMV ID assigned by TNCS
`connectionID`	Network connection ID for which an attribute value is desired (or `TNC_CONNECTIONID_ANY` to get information for the TNCS as a whole)
`attributeID`	Attribute ID for which an attribute value is desired
`bufferLength`	Length in bytes of storage referenced by `buffer` parameter (or 0 if no storage referenced)

Output Parameter	**Description**
`buffer`	Requested attribute value
`pOutValueLength`	Length in bytes of requested attribute value

Result Code	Condition
TNC_RESULT_SUCCESS	Success
TNC_RESULT_NOT_INITIALIZED	TNC_IMV_Initialize has not been called
TNC_RESULT_INVALID_PARAMETER	Invalid function parameter
TNC_RESULT_OTHER	Unspecified non-fatal error
TNC_RESULT_FATAL	Unspecified fatal error
Other result codes	Other non-fatal error

3.9.9 TNC_TNCS_SetAttribute (OPTIONAL)

```
TNC_Result TNC_TNCS_SetAttribute(
      /*in*/  TNC_IMVID imvID,
      /*in*/  TNC_ConnectionID connectionID,
      /*in*/  TNC_AttributeID attributeID,
      /*in*/  TNC_UInt32 bufferLength,
      /*in*/  TNC_BufferReference buffer);
```

Description:

An IMV calls this function to set the value of an attribute associated with a connection or with the TNCS as a whole.

This is an optional function, so the TNCS is not required to implement it. Since this function was not included in IF-IMV 1.0, many TNCSs do not implement it. IMVs MUST work properly if a TNCS does not implement this function. The IMV is never required to call this function. The TNCS MUST work with IMVs that don't call this function.The IMV SHOULD use dynamic function binding (on platforms where that is available) to determine whether the TNCS implements this function.

The IMV MUST pass its IMV ID as the `imvID` parameter, a standard or vendor-specific attribute ID as the `attributeID` parameter, and a valid network connection ID as the `connectionID` parameter. If the IMV passes a valid connection ID, the TNCS SHOULD set the attribute value for the specified connection (if supported). If the IMV passes `TNC_CONNECTIONID_ANY`, the TNCS SHOULD set the value for the TNCC (if supported). If the TNCS does not recognize the attribute ID or connection ID, it SHOULD return the `TNC_RESULT_INVALID_PARAMETER` result code. If the TNCS recognizes the attribute ID and connection ID but does not support setting an attribute value for the requested attribute ID and connection ID, the TNCS SHOULD return the `TNC_RESULT_INVALID_PARAMETER` result code.

The IMV MUST pass an attribute value in the buffer referenced by the `buffer` parameter. This attribute value SHOULD have the exact format specified in the description of the attribute ID (in section 3.6.11 for attributes defined there or in vendor-specific documentation for a vendor-specific attribute ID). The length of the attribute value MUST be exactly the number of octets (bytes) indicated by the `bufferLength` parameter. The TNCS MUST NOT modify the contents of the buffer or even access them after this function returns. Therefore, the TNCS SHOULD copy the attribute value to other storage before returning from this function.

The IMV MAY pass 0 for the `bufferLength` parameter. In this case, the IMV MUST pass NULL for the `buffer` parameter. This indicates a zero-length value for the specified attribute value.

If the IMV passes an attribute value that is not valid for the specified attribute, the TNCS MAY return the `TNC_RESULT_INVALID_PARAMETER` result code to indicate that the attribute value is

not valid. The TNCS MAY also forgo checking the validity of the attribute value and return the TNC_RESULT_SUCCESS result code but later ignore the attribute value that has been set.

Input Parameter	Description
imvID	IMV ID assigned by TNCS
connectionID	Network connection ID for which an attribute value is to be set (or TNC_CONNECTIONID_ANY to set an attribute value for the TNCS as a whole)
attributeID	Attribute ID for attribute to be set
bufferLength	Length in bytes of attribute value
Buffer	Attribute value to set

Result Code	Condition
TNC_RESULT_SUCCESS	Success
TNC_RESULT_NOT_INITIALIZED	TNC_IMV_Initialize has not been called
TNC_RESULT_INVALID_PARAMETER	Invalid function parameter
TNC_RESULT_OTHER	Unspecified non-fatal error
TNC_RESULT_FATAL	Unspecified fatal error
Other result codes	Other non-fatal error

3.9.10 TNC_TNCS_ReserveAdditionalIMVID (OPTIONAL)

```
TNC_Result TNC_TNCS_ReserveAdditionalIMVID(
      /*in*/  TNC_IMVID imvID,
      /*out*/  TNC_UInt32 *pOutIMVID);
```

Description:

An IMV calls this function to reserve an additional IMV ID for itself. This function is optional. The TNCS is not required to implement it. Since this function was not included in IF-IMV 1.0 through 1.2, many TNCSs do not support it. IMVs MUST work properly if a TNCS does not implement this function. The IMV SHOULD use dynamic function binding (on platforms where that is available) to determine whether the TNCS implements this function.

When the IF-TNCCS 2.0 [14] (and PB-TNC [21]) protocols are carrying an IF-M message, the IF-TNCCS protocol includes a header (TNCCS-IF-M-Message) housing several fields important to the processing of a received IF-M message. The IF-M Vendor ID and IF-M Subtype described in the IF-TNCCS specification are used by the TNCC and TNCS to route messages to IMC and IMV that have registered an interest in receiving messages for a particular type of component. Also present in the TNCCS-IF-M-Message header is a pair of fields that identify the IMC and IMV involved in the message exchange. The IMC and IMV Identifier fields are used for performing exclusive delivery of messages and as an indicator for correlation of received attributes. See the IF-TNCCS 2.0 protocol specification for more information on these fields.

Correlation of attributes is necessary when an IMC sends attributes describing multiple different implementations of a single type of component during an assessment; the recipient IMV(s) need to be able to determine which attributes are describing the same implementation.

For example, a single IMC might report attributes about two installed VPN implementations on the endpoint. Because the individual attributes (except the Product Information attribute) do not include an indication of which VPN product they are describing, the recipient IMV needs something to perform this correlation.Therefore, for this example, the single VPN IMC would need to obtain two IMC Identifiers from the TNC Client and consistently use one with each of the VPN implementations reported during an assessment. The VPN IMC would group all the attributes associated with a particular VPN implementation into a single IF-M message and send the message using the IMC Identifier it designates as going with the particular implementation. This approach allows the recipient IMV to recognize when attributes in future assessment messages also describe the same VPN implementation and to direct follow-up messages to the right IMC. Similarly, a single IMV may need to have multiple IMV IDs, so that an IMC can send follow-up messages to the right IMV.

The IMV MUST pass its primary IMV ID as the `imvID` parameter. The primary IMV ID is reserved by the TNCS and provided to the IMV when the `TNC_IMV_Initialize` function is called. Alternatively, the IMV can query its primary IMV ID using the `TNC_ATTRIBUTEID_PRIMARY_IMV_ID` attribute. If the IMV passes an invalid IMV ID or additional IMV ID as the `imvID` parameter instead of the primary IMV ID, the TNCS SHOULD return the `TNC_RESULT_INVALID_PARAMETER` result code. An IMV can call this function as many times it wishes. The TNCS SHOULD return a unique value every time. However, the TNCS may have a maximum number of additional IMV IDs that it supports. In that case the TNCS SHOULD return a `TNC_RESULT_OTHER` error if an IMV attempts to exceed this maximum. The TNCS is not required to reserve IMV IDs in a specific order.

The IMV MUST set `pOutIMVID` so that the TNCS can use it as an output parameter to provide the reserved additional IMV ID. With the C binding, this would involve setting `pOutIMVID` to point to a suitable storage location. The TNCS may check if `pOutIMVID` is NULL but is not required to do so.

If the TNCS returns a result code other than `TNC_RESULT_SUCCESS`, it MUST NOT store any values via the supplied parameter. But if it returns `TNC_RESULT_SUCCESS`, it MUST provide an IMV ID that has now been reserved for this IMV.

The TNCS MUST NOT modify the value stored in the `pOutIMVID` parameter after returning from this function.

Input Parameter	**Description**
`imvID`	Primary IMV ID assigned by TNCS

Output Parameter	**Description**
`pOutIMVID`	Requested additional IMV ID

Result Code	**Condition**
`TNC_RESULT_SUCCESS`	Success
`TNC_RESULT_NOT_INITIALIZED`	`TNC_IMV_Initialize` has not been called
`TNC_RESULT_INVALID_PARAMETER`	Invalid function parameter
`TNC_RESULT_OTHER`	Unspecified non-fatal error
`TNC_RESULT_FATAL`	Unspecified fatal error

Other result codes	Other non-fatal error

4 Platform Bindings

As noted above, IF-IMV is a platform-independent API. It is designed to support almost any platform. In order to ensure compatibility within a single platform, this section defines how IF-IMV SHOULD be implemented on specific platforms.

Note that if taking the "stub" approach, then IF-IMV represents an API between a TNCS and an IMV stub DLL on the same platform, although the "actual" IMV may be located remotely on a different platform.

It is assumed that platform bindings will only be created for platforms which are appropriate to a role as servers. For example, IF-IMV bindings for handheld and 16-bit consumer operating systems will not be specified as it is assumed that there will be limited (if any) implementation of TNCSs on such systems.

4.1 Microsoft Windows DLL Platform Binding

Microsoft Windows is a popular platform with many variations. This binding is intended to support only 32- or 64-bit Windows versions (e.g., Windows NT, Windows 2000, Windows 2003, or Windows XP). It is not intended to support 16-bit Windows (Windows 3.X and Windows for Workgroups), nor is it directly intended to support Windows CE, Windows 95/98/Me, or other such versions of Windows.

Implementations on one of these platforms SHOULD use this binding when possible for maximum compatibility with other IMVs or TNC Servers on the platform. However, some languages (such as Java) cannot easily implement or load DLLs. Implementations in such a language may choose not to use this binding or may write custom code to support this binding.

4.1.1 Finding, Loading, and Unloading IMVs

The loading of IMVs is parallel to the process for loading IMCs within IF-IMC, with only minor differences in behavior. With the Microsoft Windows DLL platform binding, each IMV is implemented as a DLL. This IMV DLL may be either a "stub" IMV DLL or a full IMV; this distinction is immaterial to the operation of the API.

When the DLL is installed, it is stored in a directory that can only be accessed by privileged users. The full path of the DLL is stored in a well-known registry key (defined in section 4.1.9) that can only be changed by privileged users. The TNC Server gets the value of this key and loads the IMVs using the `LoadLibrary` system call. Then it uses the `GetProcAddress` function call to access the IMV's functions, as described in section 4.1.2. The TNCS MUST always call the `TNC_IMV_Initialize` function first. When it is done using an IMV, the TNC Server calls `TNC_IMV_Terminate` and then unloads the IMV DLL using the `FreeLibrary` system call. The TNCS **MUST** listen for changes to the well-known registry key so that it can load and unload IMVs dynamically. However, the TNCS **SHOULD** delay before making changes based on registry key changes since it is common for these changes to come in batches within a few seconds during an install process. Unlike a TNCC, a TNCS **MUST NOT** ignore such changes.

Because the TNCS uses the `LoadLibrary` system call to load the IMV, DLLs implicitly referenced by the IMV may not be found and loaded unless they are in one of the system directories. This occurs because the DLL search path used when loading the IMV does not contain the directory containing the IMV DLL. It only contains the directory containing the TNCS, system directories, etc. Two good solutions to this problem are to have the IMV explicitly load any unusual DLLs that it needs and use the `GetProcAddress` function call to access functions within those DLLs, or to have the IMV delay-load any unusual DLLs and write a delay load hook that searchs the IMV's paths for those libraries.

4.1.1.1 Use of COM Objects

The TNCS and IMVs may access features such as the system management functionality available in Windows through COM objects exposed via language independent interfaces.

However, this can cause problems. Before the TNCS or IMVs can use COM objects, the COM library needs to be initialized by the calling thread using `CoInitialize` or `CoInitializeEx`. The COM library has a set of rules for initialization. For example, it should be initialized on every thread that needs to use COM objects; initializing the COM library once in a process is not sufficient. The COM library can be initialized more than once on a thread, but it should be initialized every time with the same parameters. If the TNCS has initialized the COM library with the `COINIT_APARTMENTTHREADED` option, an IMV which requires the COM library to be initialized with the `COINIT_MULTITHREADED` option might not work correctly, unless the protections described later in this section are used. Another rule is that the COM library needs to be uninitialized as many times on a thread as it has been initialized, to allow unloading of the COM objects from the process address space.

Since the TNCS and IMVs may need to initialize the COM library in mutually inconsistent ways, an IMV should not make any assumptions about the initialization parameters used by the TNCS, or even whether the TNCS has initialized the COM library. Having an IMC invoke methods on COM objects in a thread shared with the TNCS could potentially cause problems. Since TNC server software is a multivendor environment where TNCS, IMVs, and third party libraries used by IMVs are loaded in the same process but developed by different parties, it is difficult to agree on the same initialization parameters needed to initialize the COM library. Therefore, IMVs SHOULD NOT use COM (or call libraries that assume a particular initialization for the COM library) on a thread created by the TNCC. In a future version of this specification, this recommendation may be upgraded to a requirement (MUST NOT). Instead, IMCs that need to use COM should create their own threads for this purpose. If an IMV doesn't use COM objects or call libraries that assume a particular initialization for the COM library (which most libraries don't do), the creator of that IMV can ignore this section.

4.1.2 Dynamic Function Binding

The Microsoft Windows DLL platform binding does support dynamic function binding. To determine whether an IMV function is defined, a TNC Server will pass the function name to `GetProcAddress`. If the result is `NULL`, the function is not defined. Otherwise, the function is defined and the TNCS can call it using the function pointer returned. This is common practice on Windows.

A similar mechanism is used to allow an IMV to determine whether a TNCS function is defined. In fact, this mechanism is the only way that the IMV can call a TNCS function with this platform binding. A platform-specific mandatory IMV function named `TNC_IMV_ProvideBindFunction` is defined below. For instructions on how this function is used, see its description.

IMV and TNCS functions can be implemented in and called from many languages. With C++, extern "C" should be used to ensure that C linkage conventions are used for IMV and TNCS functions exposed through this API.

4.1.3 Threading

Unlike IMCs, IMV DLLs are required to be thread-safe. The IMV DLL MAY create threads. The TNC Server MUST be thread-safe. This allows the IMV DLL to do work in background threads and call the TNC Server when a recommendation is ready (for instance).

4.1.4 Platform-Specific Bindings for Basic Types

With the Microsoft Windows DLL platform binding, the basic data types defined in the IF-IMV abstract API are mapped as follows:

```
typedef unsigned long TNC_UInt32;
```

The TNC_UInt32 type is mapped to a four byte unsigned value.

```
typedef unsigned char *TNC_BufferReference;
```

The TNC_BufferReference type is mapped to a pointer. The value NULL is allowed for a TNC_BufferReference only where explicitly permitted in this specification.

4.1.5 Platform-Specific Bindings for Derived Types

With the Microsoft Windows DLL platform binding, the platform-specific derived data types defined in the IF-IMV abstract API are mapped as follows:

```
typedef TNC_MessageType *TNC_MessageTypeList;

typedef TNC_VendorID *TNC_VendorIDList;

typedef TNC_MessageSubtype *TNC_MessageSubtypeList;
```

The TNC_MessageTypeList, TNC_VendorIDList, and TNC_MessageSubtypeList types are mapped to a pointer. The value NULL is allowed for these types only where explicitly permitted in this specification.

4.1.6 Additional Platform-Specific Derived Types

The Microsoft Windows DLL platform binding for the IF-IMV API defines several additional derived data types.

4.1.6.1 Function Pointers

Function pointer types are defined for all the functions contained in the abstract API and platform binding. This makes it easy to cast function pointers returned by GetProcAddress or TNC_TNCS_BindFunction to the right type and ensure that the compiler performs type checking on arguments.

```
typedef TNC_Result (*TNC_IMV_InitializePointer)(
    TNC_IMVID imvID,
    TNC_Version minVersion,
    TNC_Version maxVersion,
    TNC_Version *pOutActualVersion);

typedef TNC_Result (*TNC_IMV_NotifyConnectionChangePointer)(
    TNC_IMVID imvID,
    TNC_ConnectionID connectionID,
    TNC_ConnectionStatus newStatus);

typedef TNC_Result (*TNC_IMV_ReceiveMessagePointer)(
    TNC_IMVID imvID,
    TNC_ConnectionID connectionID,
    TNC_BufferReference message,
    TNC_UInt32 messageLength,
    TNC_MessageType messageType);

typedef TNC_Result (*TNC_IMV_ReceiveMessageSOHPointer)(
    TNC_IMVID imvID,
    TNC_ConnectionID connectionID,
    TNC_BufferReference sohReportEntry,
    TNC_UInt32 sohRELength,
    TNC_MessageType systemHealthID);

typedef TNC_Result (*TNC_IMV_ReceiveMessageLongPointer)(
    TNC_IMVID imvID,
    TNC_ConnectionID connectionID,
    TNC_UInt32 messageFlags,
    TNC_BufferReference message,
    TNC_UInt32 messageLength,
    TNC_VendorID messageVendorID,
```

```
    TNC_MessageSubtype messageSubtype,
    TNC_UInt32 sourceIMCID,
    TNC_UInt32 destinationIMVID);

typedef TNC_Result (*TNC_IMV_SolicitRecommendationPointer)(
    TNC_IMVID imvID,
    TNC_ConnectionID connectionID);

typedef TNC_Result (*TNC_IMV_BatchEndingPointer)(
    TNC_IMVID imvID,
    TNC_ConnectionID connectionID);

typedef TNC_Result (*TNC_IMV_TerminatePointer)(
    TNC_IMVID imvID);

typedef TNC_Result (*TNC_TNCS_ReportMessageTypesPointer)(
    TNC_IMVID imvID,
    TNC_MessageTypeList supportedTypes,
    TNC_UInt32 typeCount);

typedef TNC_Result (*TNC_TNCS_ReportMessageTypesLongPointer)(
    TNC_IMVID imvID,
    TNC_VendorIDList supportedVendorIDs,
    TNC_MessageSubtypeList supportedSubtypes,
    TNC_UInt32 typeCount);

typedef TNC_Result (*TNC_TNCS_SendMessagePointer)(
    TNC_IMVID imvID,
    TNC_ConnectionID connectionID,
    TNC_BufferReference message,
    TNC_UInt32 messageLength,
    TNC_MessageType messageType);

typedef TNC_Result (*TNC_TNCS_SendMessageSOHPointer)(
    TNC_IMVID imvID,
    TNC_ConnectionID connectionID,
    TNC_BufferReference sohrReportEntry,
    TNC_UInt32 sohrRELength);

typedef TNC_Result (*TNC_TNCS_SendMessageLongPointer)(
    TNC_IMVID imvID,
    TNC_ConnectionID connectionID,
    TNC_UInt32 messageFlags,
    TNC_BufferReference message,
    TNC_UInt32 messageLength,
    TNC_VendorID messageVendorID,
    TNC_MessageSubtype messageSubtype,
    TNC_UInt32 destinationIMCID);

typedef TNC_Result (*TNC_TNCS_RequestHandshakeRetryPointer)(
    TNC_IMVID imvID,
    TNC_ConnectionID connectionID,
    TNC_RetryReason reason);

typedef TNC_Result (*TNC_TNCS_ProvideRecommendationPointer)(
    TNC_IMVID imvID,
    TNC_ConnectionID connectionID,
    TNC_IMV_Action_Recommendation recommendation);

typedef TNC_Result (*TNC_TNCS_GetAttributePointer)(
    TNC_IMVID imvID,
    TNC_ConnectionID connectionID,
```

```
    TNC_AttributeID attributeID,
    TNC_UInt32 bufferLength,
    TNC_BufferReference buffer,
    TNC_UInt32 *pOutValueLength);

typedef TNC_Result (*TNC_TNCS_SetAttributePointer)(
    TNC_IMVID imvID,
    TNC_ConnectionID connectionID,

    TNC_AttributeID attributeID,
    TNC_UInt32 bufferLength,
    TNC_BufferReference buffer);

typedef TNC_Result (*TNC_TNCS_ReserveAdditionalIMVIDPointer)(
    TNC_IMVID imvID,
    TNC_UInt32 *pOutIMVID);

typedef TNC_Result (*TNC_TNCS_BindFunctionPointer)(
    TNC_IMVID imvID,
    char *functionName,
    void **pOutfunctionPointer);

typedef TNC_Result (*TNC_IMV_ProvideBindFunctionPointer)(
    TNC_IMVID imvID,
    TNC_TNCS_BindFunctionPointer bindFunction);
```

4.1.7 Platform-Specific IMV Functions

The Microsoft Windows DLL platform binding for the IF-IMV API defines one additional function that MUST be implemented by IMVs implementing this platform binding.

4.1.7.1 TNC_IMV_ProvideBindFunction (MANDATORY)

```
TNC_Result TNC_IMV_ProvideBindFunction(
      /*in*/  TNC_IMVID imvID,
      /*in*/  TNC_TNCS_BindFunctionPointer bindFunction);
```

Description:

IMVs implementing the Microsoft Windows DLL platform binding MUST define this additional platform-specific function. The TNC Server MUST call the function immediately after calling `TNC_IMV_Initialize` to provide a pointer to the TNCS bind function. The IMV can then use the TNCS bind function to obtain pointers to any other TNCS functions.

In the `imvID` parameter, the TNCS MUST pass the value provided to `TNC_IMV_Initialize`. In the `bindFunction` parameter, the TNCS MUST pass a pointer to the TNCS bind function. IMVs MAY check if `imvID` matches the value previously passed to `TNC_IMV_Initialize` and return `TNC_RESULT_INVALID_PARAMETER` if not, but they are not required to make this check.

Input Parameter	Description
`imvID`	IMV ID assigned by TNCS
`bindFunction`	Pointer to `TNC_TNCS_BindFunction`

Error Code	Condition
`TNC_RESULT_SUCCESS`	Success

TNC_RESULT_NOT_INITIALIZED	TNC_IMV_Initialize has not been called
TNC_RESULT_INVALID_PARAMETER	Invalid function parameter
TNC_RESULT_OTHER	Unspecified non-fatal error
TNC_RESULT_FATAL	Unspecified fatal error
Other error codes	Other non-fatal error

4.1.8 Platform-Specific TNC Server Functions

The Microsoft Windows DLL platform binding for the IF-IMV API defines one additional function that MUST be implemented by TNC Servers implementing this platform binding.

4.1.8.1 TNC_TNCS_BindFunction (MANDATORY)

```
TNC_Result TNC_TNCS_BindFunction(
      /*in*/  TNC_IMVID imvID,
      /*in*/  char *functionName,
      /*out*/ void **pOutFunctionPointer);
```

Description:

TNC Servers implementing the Microsoft Windows DLL platform binding MUST define this additional platform-specific function. An IMV can use this function to obtain pointers to other TNCS functions. To obtain a pointer to a TNCS function, an IMV calls TNC_TNCS_BindFunction. The IMV obtains a pointer to TNC_TNCS_BindFunction from TNC_IMV_ProvideBindFunction.

The IMV MUST set the imvID parameter to the IMV ID value provided to TNC_IMV_Initialize. TNCSs MAY check if imvID matches the value previously passed to TNC_IMV_Initialize and return TNC_RESULT_INVALID_PARAMETER if not, but they are not required to make this check. The IMV MUST set the functionName parameter to a pointer to a C string identifying the function whose pointer is desired (i.e. "TNC_TNCS_SendMessage"). The IMV MUST set the pOutFunctionPointer parameter to a pointer to storage into which the desired function pointer will be stored. If the TNCS does not define the requested function, NULL MUST be stored at pOutFunctionPointer. Otherwise, a pointer to the requested function MUST be stored at pOutFunctionPointer. In either case, TNC_RESULT_SUCCESS SHOULD be returned. Once an IMV obtains a pointer to a particular function, the TNCS MUST always return the same function pointer value to that IMV for that function name. This requirement does not apply across IMV termination and reinitialization.

Input Parameter	Description
imvID	IMV ID assigned by TNCS
functionName	Name of function whose pointer is requested

Output Parameter	Description
pOutFunctionPointer	Requested function pointer

Error Code	Condition
TNC_RESULT_SUCCESS	Success

TNC_RESULT_INVALID_PARAMETER	Invalid function parameter
TNC_RESULT_OTHER	Unspecified non-fatal error
TNC_RESULT_FATAL	Unspecified fatal error
Other error codes	Other non-fatal error

4.1.9 Well-known Registry Key

As discussed above, a well-known registry key is used by the TNCS to load IMVs. For Windows platforms, this key is defined within the HKEY_LOCAL_MACHINE hive as follows. (The HKLM hive is used since there will often be no logged on user to give context for any other hive.)

- HKEY_LOCAL_MACHINE
  - → Software
  - → Trusted Computing Group
  - → TNC
  - → IMVs
  - → [Human readable name of IMV], 0..n

Each IMV key contains an (unordered) set of values, as follows:

- the value "*Path"* is a REG_SZ String which contains the fully qualified path to an IMV DLL to be loaded.
- the optional value "*Description"* is a REG_SZ String which contains a vendor-specific human-readable description of the IMV DLL

The name and description are for ease of administration and may be ignored by the TNCS, except for human interface purposes; only the Path data matters. Duplicate paths are OK. Additional values or keys may be present within the keys listed above. TNC Servers and IMVs MUST ignore unrecognized values and keys.

An extension mechanism has been defined so that vendors can place vendor-specific keys or values in the TNC key or any subkey without risking name collisions. The name of such a vendor-specific key or value must begin with the vendor ID (as defined in section 3.2.3) of the vendor who defined this extension. The vendor ID must be immediately followed in the name by an underscore which may be followed by any string.

The manner in which these vendor-specific values are used is up to the vendor that defines such a value. For instance, a TNC Server vendor with vendor ID 2 could specify that any IMV can populate its key at install time with a value named 2_SupportPhone and that vendor's TNC Server can read this value and display it in the TNC Server's status panel next to the IMV name. The only requirement, as stated above, is that TNC Servers and IMVs MUST ignore unrecognized values and keys.

4.2 UNIX/Linux Dynamic Linkage Platform Binding

UNIX and Linux operating systems are used for servers, desktops, and even embedded devices. There are hundreds of varieties of UNIX and Linux dating back to the 1970s. One platform binding cannot support them all. However, this binding supports all varieties of Linux that conform to the Linux Standard Base 1.0.0 or later and all varieties of UNIX that conform to UNIX 98 or any version of the Single UNIX Specification. This includes most varieties of UNIX and Linux currently in use.

Implementations on one of these platforms SHOULD use this binding when possible for maximum compatibility with other IMVs and TNC Servers on the platform. However, some

languages (such as Java) cannot easily implement or load shared libraries. Implementations in such a language may choose not to use this binding or to write custom code to support this binding.

4.2.1 Finding, Loading, and Unloading IMVs

With the UNIX/Linux Dynamic Linkage platform binding, each IMV is implemented as a dynamically loaded executable file (also known as a shared object or DLL). When the IMV is installed, its executable file should be stored in a directory that can only be accessed by privileged users. Then an entry is created in the `/etc/tnc_config` file that gives the full path of the executable file. See section 4.2.3 for details on the format of this file.

The TNC Server opens the `/etc/tnc_config` file, reads the entries in the file, and determines which of them should be loaded (using optional local configuration). For each IMV to be loaded, the TNC Server passes the full path of the executable file to the `dlopen` system call. The value passed as the `mode` parameter to the `dlopen` system call is platform-specific and not specified here. The TNC Server uses the `dlsym` function call to access the IMV's functions, as described in section 4.2.2. The TNCS MUST always call the `TNC_IMV_Initialize` function first. When it is done using an IMV, the TNC Server calls `TNC_IMV_Terminate` and then unloads the IMV executable file using the `dlclose` system call.

If the TNCS receives a HUP signal (which may be sent with the `kill` command), the TNCS SHOULD check the `/etc/tnc_config` file for changes and load or unload IMVs as needed to match the latest list.

4.2.2 Dynamic Function Binding

The UNIX/Linux Dynamic Linkage platform binding does support dynamic function binding. To determine whether an IMV function is defined, a TNC Server will pass the function name to `dlsym`. If the result is `NULL`, the function is not defined. Otherwise, the function is defined and the TNCS can call it using the function pointer returned. This is common practice on UNIX and Linux.

A similar mechanism is used to allow an IMV to determine whether a TNCS function is defined. In fact, this mechanism is the only way that the IMV can call a TNCS function with this platform binding. A platform-specific mandatory IMV function named `TNC_IMV_ProvideBindFunction` is defined below. For instructions on how this function is used, see its description.

IMV0 and TNCS functions can be implemented in and called from many languages. With C++, extern "C" should be used to ensure that C linkage conventions are used for IMV and TNCS functions exposed through this API.

4.2.3 Format of `/etc/tnc_config`

The `/etc/tnc_config` file specifies the set of IMVs available for TNCSs to load. TNCSs are not required to load these IMVs. A TNCS may be configured to ignore this file, load any subset of the IMVs listed here, load a superset of those IMVs, or (most common) load the IMVs in the list. This provides a simple, standard way for the list of IMVs to be specified but allows TNCCs to be configured to only load a particular set of trusted IMVs.

The `/etc/tnc_config` file is a UTF-8 file. However, TNCSs are only required to support US-ASCII characters (a subset of UTF-8). If a TNCS encounters a character that is not US-ASCII and the TNCC can not process UTF-8 properly, the TNCS SHOULD indicate an error and not load the file at all. In fact, the TNCS SHOULD respond to any problem with the file by indicating an error and not loading the file at all.

All characters specified here are specified in standard Unicode notation (U+nnnn where nnnn are hexadecimal characters indicating the code points.

The `/etc/tnc_config` file is composed of zero or more lines. Each line ends in U+000A. No other control characters (characters with the Unicode category Cc) are permitted in the file.

A line that begins with U+0023 is a comment. All other characters on the line should be ignored. A line that does not contain any characters should also be ignored.

A line that begins with “IMV ” (U+0049, U+004D, U+0056, U+0020) specifies an IMV that may be loaded. The next character MUST be U+0022 (QUOTATION MARK). This MUST be followed by a human-readable IMV name (potentially zero length) and another U+0022 character (QUOTATION MARK). Of course, the IMV name cannot contain a U+0022 (QUOTATION MARK). But it can contain spaces or other characters. After the U+0022 that terminates the human-readable name MUST come a space (U+0020) and then the full path of the IMV executable file (up to but not including the U+000A that terminates the line). The path to the IMV executable file MUST NOT be a partial path.

The `/etc/tnc_config` file must not contain IMVs with the same human-readable name. An IMV that encounters such a file SHOULD indicate the error and MAY not load the file at all. It MAY also change the IMV names to make them unique. Identical full paths are permitted but the TNCC MAY ignore entries with identical paths if they will cause problems for it.

An extension mechanism has been defined so that vendors can place vendor-specific data in the `/etc/tnc_config` file without risking conflicts. A line that contains such vendor-specific data must begin with the vendor ID (as defined in section 3.2.3) of the vendor who defined this extension. The vendor ID must be immediately followed in the name by an underscore which may be followed by any string except control characters until the end of line (U+000A).

The internal format of this vendor-specific data and the manner in which it is to be used should be specified by the vendor whose vendor ID is used to define the extension. For instance, a TNCS vendor with vendor ID 2 could specify that any IMV can add a line at install time that begins with 2_SupportPhoneIMV, then the IMV’s human-readable name and the IMV vendor’s support telephone number. The defining vendor’s TNCS (or any other TNCS) can read this phone number and display it in the TNCS’s status panel next to the IMV name.

TNCSs and IMVs SHOULD ignore unrecognized vendor-specific data. This recommendation is backwards-compatible with the recommendation in IF-IMV 1.0 for TNCSs and IMVs to ignore lines in `/etc/tnc_config` with unrecognized syntax.

A line that does not match the comment, empty, imv, or vendor productions below SHOULD be ignored by a TNCS and IMVs that are using the Linux/UNIX Platform Binding unless otherwise specified by a future version of this binding. This provides for future extensions to this file format.

Here is a specification of the file format using ABNF as defined in [3].

```
tnc_config = *line
line = (comment / empty / imc / java-imc / imv / java-imv / vendor /
other) %x0A
comment = %x23 *(%x01-09 / %x0B-22 / %x24-1FFFFF)
empty = ""
imc = %x49.4D.43.20.22 name %x22.20 path
java-imc = %x4a.41.56.41.2d.49.4D.43.20.22 name %x22.20 class %x20 path
imv = %x49.4D.56.20.22 name %x22.20 path
java-imv = %x4a.41.56.41.2d.49.4D.56.20.22 name %x22.20 class %x20 path
name = *(%x01-09 / %x0B-21 / %x23-1FFFFF)
class = *(%x01-09 / %x0B-1F / %x21-1FFFFF)
path = *(%x01-09 / %x0B-1FFFFF)
digit = (%x30-39)
vendor = *digit %x5f *(%x01-09 / %x0B-1FFFFF)
other = 1*(%x01-09 / %x0B-1FFFFF) ; But match more specific rules first
```

Note that lines that match the `imc`, `java-imc`, and `java-imv` productions are ignored for the purposes of the Linux/UNIX Platform Binding for IF-IMV. Note also that the `other` production is only employed if no other production matches a line.

Here is a sample file specifying one IMV named “AV” located at /usr/bin/myav/av.so.

```
# Simple TNC config file

IMV "AV" /usr/bin/myav/av.so
```

4.2.4 Threading

Unlike IMC’s, IMV executable files are required to be thread-safe. The IMV MAY create threads. The TNC Server MUST be thread-safe. This allows the IMV DLL to do work in background threads and call the TNC Server when messages are ready to send (for instance). Both the IMV and the TNC Server MUST use POSIX threads (pthreads) for threading and synchronization to ensure compatibility.

4.2.5 Platform-Specific Bindings for Basic Types

With the UNIX/Linux Dynamic Linkage platform binding, the basic data types defined in the IF-IMV abstract API are mapped as follows:

```
typedef unsigned long TNC_UInt32;
```

The TNC_UInt32 type is mapped to a four byte unsigned value.

```
typedef unsigned char *TNC_BufferReference;
```

The `TNC_BufferReference` type is mapped to a pointer. The value `NULL` is allowed for a `TNC_BufferReference` only where explicitly permitted in this specification.

4.2.6 Platform-Specific Bindings for Derived Types

With the UNIX/Linux Dynamic Linkage platform binding, the platform-specific derived data types defined in the IF-IMV abstract API are mapped as follows:

```
typedef TNC_MessageType *TNC_MessageTypeList;

typedef TNC_VendorID *TNC_VendorIDList;

typedef TNC_MessageSubtype *TNC_MessageSubtypeList;
```

The `TNC_MessageTypeList`, `TNC_VendorIDList`, and `TNC_MessageSubtypeList` types are mapped to a pointer. The value `NULL` is allowed for these types only where explicitly permitted in this specification.

4.2.7 Additional Platform-Specific Derived Types

The UNIX/Linux Dynamic Linkage DLL platform binding for the IF-IMV API defines several additional derived data types.

4.2.7.1 Function Pointers

Function pointer types are defined for all the functions contained in the abstract API and platform binding. This makes it easy to cast function pointers returned by `dlsym` or `TNC_TNCS_BindFunction` to the right type and ensure that the compiler performs type checking on arguments.

```
typedef TNC_Result (*TNC_IMV_InitializePointer)(
    TNC_IMVID imvID,
    TNC_Version minVersion,
    TNC_Version maxVersion,
    TNC_Version *pOutActualVersion);
```

```
typedef TNC_Result (*TNC_IMV_NotifyConnectionChangePointer)(
    TNC_IMVID imvID,
    TNC_ConnectionID connectionID,
    TNC_ConnectionStatus newStatus);

typedef TNC_Result (*TNC_IMV_ReceiveMessagePointer)(
    TNC_IMVID imvID,
    TNC_ConnectionID connectionID,
    TNC_BufferReference message,
    TNC_UInt32 messageLength,
    TNC_MessageType messageType);

typedef TNC_Result (*TNC_IMV_ReceiveMessageSOHPointer)(
    TNC_IMVID imvID,
    TNC_ConnectionID connectionID,
    TNC_BufferReference sohReportEntry,
    TNC_UInt32 sohRELength,
    TNC_MessageType systemHealthID);
typedef TNC_Result (*TNC_IMV_ReceiveMessageLongPointer)(
    TNC_IMVID imvID,
    TNC_ConnectionID connectionID,
    TNC_UInt32 messageFlags,
    TNC_BufferReference message,
    TNC_UInt32 messageLength,
    TNC_VendorID messageVendorID,
    TNC_MessageSubtype messageSubtype,
    TNC_UInt32 sourceIMCID,
    TNC_UInt32 destinationIMVID);

typedef TNC_Result (*TNC_IMV_SolicitRecommendationPointer)(
    TNC_IMVID imvID,
    TNC_ConnectionID connectionID);

typedef TNC_Result (*TNC_IMV_BatchEndingPointer)(
    TNC_IMVID imvID,
    TNC_ConnectionID connectionID);

typedef TNC_Result (*TNC_IMV_TerminatePointer)(
    TNC_IMVID imvID);

typedef TNC_Result (*TNC_TNCS_ReportMessageTypesPointer)(
    TNC_IMVID imvID,
    TNC_MessageTypeList supportedTypes,
    TNC_UInt32 typeCount);

typedef TNC_Result (*TNC_TNCS_ReportMessageTypesLongPointer)(
    TNC_IMVID imvID,
    TNC_VendorIDList supportedVendorIDs,
    TNC_MessageSubtypeList supportedSubtypes,
    TNC_UInt32 typeCount);

typedef TNC_Result (*TNC_TNCS_SendMessagePointer)(
    TNC_IMVID imvID,
    TNC_ConnectionID connectionID,
    TNC_BufferReference message,
    TNC_UInt32 messageLength,
    TNC_MessageType messageType);

typedef TNC_Result (*TNC_TNCS_SendMessageSOHPointer)(
    TNC_IMVID imvID,
    TNC_ConnectionID connectionID,
```

```
    TNC_BufferReference sohrReportEntry,
    TNC_UInt32 sohrRELength);
typedef TNC_Result (*TNC_TNCS_SendMessageLongPointer)(
    TNC_IMVID imvID,
    TNC_ConnectionID connectionID,
    TNC_UInt32 messageFlags,
    TNC_BufferReference message,
    TNC_UInt32 messageLength,
    TNC_VendorID messageVendorID,
    TNC_MessageSubtype messageSubtype,
    TNC_UInt32 destinationIMCID);

typedef TNC_Result (*TNC_TNCS_RequestHandshakeRetryPointer)(
    TNC_IMVID imvID,
    TNC_ConnectionID connectionID,
    TNC_RetryReason reason);

typedef TNC_Result (*TNC_TNCS_ProvideRecommendationPointer)(
    TNC_IMVID imvID,
    TNC_ConnectionID connectionID,
    TNC_IMV_Action_Recommendation recommendation);

typedef TNC_Result (*TNC_TNCS_GetAttributePointer)(
    TNC_IMVID imvID,
    TNC_ConnectionID connectionID,

    TNC AttributeID, attributeID,
    TNC_UInt32 bufferLength,
    TNC_BufferReference buffer,
    TNC_UInt32 *pOutValueLength);

typedef TNC_Result (*TNC_TNCS_SetAttributePointer)(
    TNC_IMVID imvID,
    TNC_ConnectionID connectionID,

    TNC_AttributeID attributeID,
    TNC_UInt32 bufferLength,
    TNC_BufferReference buffer);

typedef TNC_Result (*TNC_TNCS_ReserveAdditionalIMVIDPointer)(
    TNC_IMVID imvID,
    TNC_UInt32 *pOutIMVID);

typedef TNC_Result (*TNC_TNCS_BindFunctionPointer)(
    TNC_IMVID imvID,
    char *functionName,
    void **pOutfunctionPointer);

typedef TNC_Result (*TNC_IMV_ProvideBindFunctionPointer)(
    TNC_IMVID imvID,
    TNC_TNCS_BindFunctionPointer bindFunction);
```

4.2.8 Platform-Specific IMV Functions

The UNIX/Linux Dynamic Linkage platform binding for the IF-IMV API defines one additional function that MUST be implemented by IMVs implementing this platform binding.

4.2.8.1 TNC_IMV_ProvideBindFunction (MANDATORY)

```
TNC_Result TNC_IMV_ProvideBindFunction(
      /*in*/  TNC_IMVID imvID,
      /*in*/  TNC_TNCS_BindFunctionPointer bindFunction);
```

Description:

IMVs implementing the UNIX/Linux Dynamic Linkage platform binding MUST define this additional platform-specific function. The TNC Server MUST call the function immediately after calling `TNC_IMV_Initialize` to provide a pointer to the TNCS bind function. The IMV can then use the TNCS bind function to obtain pointers to any other TNCS functions.

In the `imvID` parameter, the TNCS MUST pass the value provided to `TNC_IMV_Initialize`. In the `bindFunction` parameter, the TNCS MUST pass a pointer to the TNCS bind function. IMVs MAY check if `imvID` matches the value previously passed to `TNC_IMV_Initialize` and return `TNC_RESULT_INVALID_PARAMETER` if not, but they are not required to make this check.

Input Parameter	Description
`imvID`	IMV ID assigned by TNCS
`bindFunction`	Pointer to `TNC_TNCS_BindFunction`

Error Code	Condition
`TNC_RESULT_SUCCESS`	Success
`TNC_RESULT_NOT_INITIALIZED`	`TNC_IMV_Initialize` has not been called
`TNC_RESULT_INVALID_PARAMETER`	Invalid function parameter
`TNC_RESULT_OTHER`	Unspecified non-fatal error
`TNC_RESULT_FATAL`	Unspecified fatal error
Other error codes	Other non-fatal error

4.2.9 Platform-Specific TNC Server Functions

The UNIX/Linux Dynamic Linkage platform binding for the IF-IMV API defines one additional function that MUST be implemented by TNC Servers implementing this platform binding.

4.2.9.1 TNC_TNCS_BindFunction (MANDATORY)

```
TNC_Result TNC_TNCS_BindFunction(
      /*in*/  TNC_IMVID imvID,
      /*in*/  char *functionName,
      /*out*/ void **pOutFunctionPointer);
```

Description:

TNC Servers implementing the UNIX/Linux Dynamic Linkage platform binding MUST define this additional platform-specific function. An IMV can use this function to obtain pointers to other TNCS functions. To obtain a pointer to a TNCS function, an IMV calls `TNC_TNCS_BindFunction`. The IMV obtains a pointer to `TNC_TNCS_BindFunction` from `TNC_IMV_ProvideBindFunction`.

The IMV MUST set the `imvID` parameter to the IMV ID value provided to `TNC_IMV_Initialize`. TNCSs MAY check if `imvID` matches the value previously passed to `TNC_IMV_Initialize` and return `TNC_RESULT_INVALID_PARAMETER` if not, but they are not

required to make this check. The IMV MUST set the functionName parameter to a pointer to a C string identifying the function whose pointer is desired (i.e. "TNC_TNCS_SendMessage"). The IMV MUST set the pOutFunctionPointer parameter to a pointer to storage into which the desired function pointer will be stored. If the TNCS does not define the requested function, NULL MUST be stored at pOutFunctionPointer. Otherwise, a pointer to the requested function MUST be stored at pOutFunctionPointer. In either case, TNC_RESULT_SUCCESS SHOULD be returned.

Input Parameter	Description
imvID	IMV ID assigned by TNCS
functionName	Name of function whose pointer is requested

Output Parameter	Description
pOutFunctionPointer	Requested function pointer

Error Code	Condition
TNC_RESULT_SUCCESS	Success
TNC_RESULT_INVALID_PARAMETER	Invalid function parameter
TNC_RESULT_OTHER	Unspecified non-fatal error
TNC_RESULT_FATAL	Unspecified fatal error
Other error codes	Other non-fatal error

4.3 Java Platform Binding

The Java Platform provides remarkable portability, allowing the same code to run on many operating systems. It also includes features that allow partially trusted code to be loaded with reduced privileges. There is a desire to support IF-IMV on the Java Platform, especially to support loading IMVs with reduced privileges to address security and reliability concerns. Therefore, the Java Platform Binding for IF-IMV has been developed.

At this time, the only versions of the Java Platform that are supported with the Java Platform Binding for IF-IMV are the Java 2 Platform Standard Edition versions 1.4.2 and later. However, other versions may be supported at a later time. Implementations of the IF-IMV specification on the Java 2 Platform SHOULD use this binding when possible for maximum compatibility with other IMCs and TNC Clients on the platform.

4.3.1 Object Orientation

The Java Platform Binding for IF-IMV is designed to take advantage of the Java Platform's support for object orientation. Three Java interfaces have been defined that correspond to the kinds of objects inherent in the IF-IMV Abstract API: IMV, TNCS, and IMVConnection. The functions described in the IF-IMV Abstract API have been mapped to methods in these interfaces. Interfaces were used instead of classes to leave implementers the freedom to use whatever class hierarchy they need or want. An additional TNCConstants interface has been defined to contain constant values shared between IF-IMC and IF-IMV.

All IMVs that implement the Java Platform Binding for IF-IMV MUST implement the IMV interface. All TNCSs that implement the Java Platform Binding for IF-IMV MUST implement the TNCS interface and provide objects that implement the IMVConnection interface as needed. The TNCS MUST also define all the interfaces in the IF-IMV API.

4.3.2 Exception Handling

The exception handling capabilities of the Java Platform provide greater robustness than the result code mechanism used by the IF-IMV Abstract API since exceptions must be explicitly ignored while result codes are ignored by default. Therefore, the Java Platform Binding for IF-IMV defines a `TNCException` class that wraps the result codes defined in the IF-IMV Abstract API. This class MUST be defined by all TNCSs.

4.3.3 Limited Privileges

The Java Platform has always included support for running code with limited privileges. With the Java 2 Platform (JDK 1.2 and later), this is implemented with AccessControllers, Permissions, and other components of the Java 2 Platform security architecture. These features are useful for IF-IMV, where it may be desirable to load and run a partially trusted IMV.

The Java Platform Binding for IF-IMV does not define any new Permissions. All IMVs loaded by the TNCS are assumed to be trusted to call any TNCS methods and vice versa. However, this does not mean that the TNCS and IMVs should completely trust each other.

The Java 2 Platform does define many Permissions. Many system methods check to ensure that calling code holds those permissions before allowing access. Since a TNCS may have more or fewer privileges than an IMV if the TNCS and IMV were loaded from different CodeSources, a TNCS and IMV cannot trust each other.

When a TNCS loads an IMV from any external source (one that is not delivered with the TNCS), it MUST use a class loader that will determine and assign the appropriate permissions (such as the URLClassLoader).

When an IMV calls a method of a class included in the TNCS, the TNCS code MUST recognize that the IMC's permissions may be much less than those of the TNCS. The code in the TNCS's called method will run with the intersection of the IMV's permissions and the TNCS's. To perform privileged operations, the TNCS's code MAY use a doPrivileged method to regain its normal permissions and perform privileged actions. Alternatively, the TNCS's code MAY queue data for later processing by code with more permissions. In either case, the TNCS's code MUST check the IMV's request and the arguments supplied very carefully. The IMV's code MUST NOT be trusted unless the TNCS knows that the IMV's privileges are as great as the TNCS's (as when the IMV was loaded from the same CodeSource as the TNCS).

Likewise, when the TNCS calls a method of an IMV, the IMV code MUST recognize that the TNCS's permissions may be much less than those of the IMV. The code in the IMV's called method will run with the intersection of the IMV's permissions and the TNCS's. To perform privileged operations, the IMV's code MAY use a doPrivileged method to regain its normal permissions and perform privileged actions. Alternatively, the IMV's code MAY queue data for later processing by code with more permissions. In either case, the IMV's code MUST check the TNCS's call and the arguments supplied very carefully. The TNCS's code MUST NOT be trusted. The IMV MUST regard IF-M messages as untrusted unless the IMV can authenticate them in some manner or the IMV determines that the TNCS can be trusted enough to perform the operations requested by the IF-M messages. The simplest way to meet this last criterion is for the IMV to only perform operations triggered by IMV messages in its receiveMessage method and to do so without using doPrivileged. This will ensure that the available Permissions are the intersection of the IMV's and TNCS's permissions so the IMV will not accidentally perform any operations that the TNCS is not already trusted to perform.

4.3.4 Finding, Loading, and Unloading IMVs

With the Java platform binding, each IMV is implemented as a jar file. When the IMV is installed and is intended to be usable by any TNCS on the system, its jar file SHOULD be stored in a directory that can be read by any user but can only be modified by privileged users. Then a JAVA-IMV entry SHOULD BE created in the `tnc_config` file giving the full path of the jar file. The privileges of the jar file and the `tnc_config` file should be set so that they can be read by

any user but can only be modified by privileged users. See section 4.3.6 for details on the format of the `tnc_config` file.

A TNC Server that wishes to load a Java IMV SHOULD open the `tnc_config` file on the system, read the JAVA-IMV entries in the file, and determine which of them should be loaded (using optional local configuration or any other algorithm). For each IMV to be loaded, the TNC Server SHOULD create a new instance of the IMV class, using the full path of the jar file and the class name for the IMV class as specified in the `tnc_config` file to load the class and call the noargs constructor for that class. When loading an IMV class in this manner, the TNCS MUST use a class loader that will determine and assign the appropriate permissions (such as the URLClassLoader).

The TNCS MUST always call the IMV's `initialize` method first. When it is finished with an IMV, the TNC Server MUST call the IMV's `terminate` method.

As described in the Security Considerations section, loading an IMV into the same JVM as the TNCS can compromise the TNCS and other IMVs if the IMV is later found to be untrustworthy. Also, an unstable IMV can crash the whole TNCS. However, the risk of this is considerably less with the Java Platform Binding than with the Windows DLL Binding, especially if the Permissions assigned to the IMV are minimal.

4.3.5 Dynamic Function Binding

The Java Platform Binding for IF-IMV does support dynamic function binding. Thus to allow a TNCS or IMV to define methods that go beyond those included in this Abstract API and allow the other party to determine whether the Abstract API optional methods are implemented two techniques are used.

For a TNC Server to determine whether an optional IMV method is implemented the TNC Server should make a call to the method. If the method is not implemented, an `UnsupportedOperationException` is thrown. The same mechanism is employed to handle optional TNCS methods.

Extensions to the IF-IMV API (for new versions of the IF-IMV API and vendor extensions) are handled using interfaces. To define an extension to the IF-IMV API (e.g. adding SOH support as done below), the party defining the extension (TCG or a vendor) must place the new methods and fields in a new interface with an appropriate name. Code that wishes to use an extension with a particular IMC or TNCC must first check whether that IMV or TNCS implements the interface defined for the extension using the `instanceof` operator. If the interface is implemented, the code casts the IMC or TNCC object to the interface type and uses the new methods and fields.

For vendor-specific extensions to the IF-IMV API, the name of the new interface must begin with "TNC_XXX_" where XXX is the vendor ID of the vendor defining the extension. This will help avoid name collisions and clarify where the vendor extension came from.

4.3.6 Format of the tnc_config file

The `tnc_config` file specifies the set of IMVs available for TNCSs to load. TNCSs are not required to load these IMVs. A TNCS may be configured to ignore this file, load any subset of the IMVs listed here, load a superset of those IMVs, or (most common) load the IMVs in the list. This provides a simple, standard way for the list of IMVs to be specified but allows TNCSs to be configured to only load a particular set of trusted IMVs.

The `tnc_config` file is a UTF-8 file. However, TNCSs are only required to support US-ASCII characters (a subset of UTF-8). If a TNCS encounters a character that is not US-ASCII and the TNCS can not process UTF-8 properly, the TNCS SHOULD indicate an error and not load the file at all. In fact, the TNCS SHOULD respond to any problem with the file by indicating an error and not loading the file at all.

All characters specified here are specified in standard Unicode notation (U+nnnn where nnnn are hexadecimal characters indicating the code points.

The `tnc_config` file is composed of zero or more lines. Each line ends in U+000A. No other control characters (characters with the Unicode category Cc) are permitted in the file.

A line that begins with U+0023 is a comment. All other characters on the line should be ignored. A line that does not contain any characters should also be ignored.

A line that begins with "JAVA-IMV " (U+004A, U+0041, U+0056, U+0041, U+002D, U+0049, U+004D, U+0043, U+0020) specifies a Java IMV (an IMV that uses the Java Platform Binding) that may be loaded. The next character MUST be U+0022 (QUOTATION MARK). This MUST be followed by a human-readable IMV name (potentially zero length) and another U+0022 character (QUOTATION MARK). Of course, the human-readable IMV name cannot contain a U+0022 (QUOTATION MARK). But it can contain spaces or other characters. After the U+0022 that terminates the human-readable name MUST come a space (U+0020), the fully qualified class name of the IMV class (which MUST NOT include a space), followed by a space (U+0020). After this space MUST come the full path of the IMV jar file (which runs up to but does not include the U+000A that terminates the line). The path to the IMV jar file MUST NOT be a partial path. For maximum compatibility, the fully qualified class name SHOULD NOT contain any characters that are not US-ASCII characters.

The `tnc_config` file must not contain more than one Java IMV with the same human-readable name. A TNCS that encounters such a file SHOULD indicate the error and MAY not load the file at all. It MAY also change the IMV names to make them unique. Identical class names and full paths are permitted but the TNCS MAY ignore entries with identical class names or paths if they will cause problems for it.

An extension mechanism has been defined so that vendors can place vendor-specific data in the `tnc_config` file without risking conflicts. A line that contains such vendor-specific data must begin with the vendor ID (as defined in section 3.2.3) of the vendor who defined this extension. The vendor ID must be immediately followed in the name by an underscore which may be followed by any string except control characters until the end of line (U+000A).

The internal format of this vendor-specific data and the manner in which it is to be used should be specified by the vendor whose vendor ID is used to define the extension. For instance, a TNCS vendor with vendor ID 2 could specify that any IMV can add a line at install time that begins with 2_SupportPhoneIMV, then the IMV's human-readable name and the IMV vendor's support telephone number. The defining vendor's TNCS (or any other TNCS) can read this phone number and display it in the TNCS's status panel next to the IMV name.

TNCSs and IMVs SHOULD ignore unrecognized vendor-specific data. This recommendation is backwards-compatible with the recommendation in IF-IMV 1.0 for TNCSs and IMVs to ignore lines in the `tnc_config` file with unrecognized syntax.

A line that does not match the comment, empty, java-imv, or vendor productions below SHOULD be ignored by the TNCS and IMVs unless otherwise specified by a future version of this binding. This provides for future extensions to this file format.

Here is a specification of the file format using ABNF as defined in [3].

```
tnc_config = *line
line = (comment / empty / imc / java-imc / imv / java-imv / vendor /
other) %x0A
comment = %x23 *(%x01-09 / %x0B-22 / %x24-1FFFFF)
empty = ""
imc = %x49.4D.43.20.22 name %x22.20 path
java-imc = %x4a.41.56.41.2d.49.4D.43.20.22 name %x22.20 class %x20 path
imv = %x49.4D.56.20.22 name %x22.20 path
java-imv = %x4a.41.56.41.2d.49.4D.56.20.22 name %x22.20 class %x20 path
```

```
name = *(%x01-09 / %x0B-21 / %x23-1FFFFF)
class = *(%x01-09 / %x0B-1F / %x21-1FFFFF)
path = *(%x01-09 / %x0B-1FFFFF)
digit = (%x30-39)
vendor = *digit %x5f *(%x01-09 / %x0B-1FFFFF)
other = 1*(%x01-09 / %x0B-1FFFFF) ; But match more specific rules first
```

Note that lines that match the `imc`, `imv`, and `java-imc` productions are ignored for the purposes of the Java Platform Binding for IF-IMV. Note also that the `other` production is only employed if no other production matches a line.

Here is a sample file specifying one Java IMV named “Example IMV” with a fully qualified class name of com.example.ExampleIIMV and a jar file path of /usr/bin/example_imv.jar.

```
# Simple Java IMV config file

JAVA-IMV "Example IMV" com.example.ExampleIMV /usr/bin/example_imv.jar
```

4.3.7 Location of the tnc_config file

The location of the `tnc_config` file depends on the operating system in use. For Windows operating systems, the file SHOULD go in the C:\WINDOWS directory. For Linux and UNIX and MacOS X operating systems, the file SHOULD go in the /etc directory. This specification does not define a standard location for the `tnc_config` file on other operating systems at this time. IMVs and TNCSs MAY use the os.name property to determine which operating system they are running on and choose the appropriate location for the `tnc_config` file. If the value of the os.name property begins with “Windows”, then the operating system is probably a Windows operating system. If not, it’s probably a Linux, UNIX, or MacOS X operating system.

If the directory described above does not exist, the IMV or TNCS should not create it. These directories are a basic part of the Windows and Linux/UNIX/MacOS X operating systems. If they do not exist, there is some problem that will probably require administrative intervention.

4.3.8 Threading

With the Java Binding for IF-IMV, IMVs are required to be thread-safe. An IMV MAY create threads. The TNC Server MUST be thread-safe. This allows the IMV DLL to do work in background threads and call the TNC Server when messages are ready to send (for instance).

4.3.9 Attributes

Instead of representing attribute values with a byte array, the Java Platform Binding for IF-IMV uses objects. For each attribute ID defined in section 3.6.11, this section explains the object used by the Java Platform Binding for IF-IMV to represent values for that attribute.

4.3.9.1 Preferred Language Attribute

The Preferred Language attribute indicates which human-readable language(s) are preferred for a particular connection or for a TNCS as a whole.

With the Java Platform Binding for IF-IMV, attribute values for the Preferred Language attribute are represented as a String containing an Accept-Language header as defined in IETF RFC 3282 [3] (US-ASCII only, no control characters allowed). This header lists the languages preferred for human-readable messages. If no language preference information is available, a zero length string is used. The string is not terminated by a NUL character since this convention is not employed in the Java programming language. Thus, the Java Platform Binding for IF-IMV uses essentially the same attribute value as the abstract binding but translates it into the types and classes native to the Java platform.

4.3.9.2 Reason String Attribute

The Reason String attribute allows an IMV to deliver to the TNCS a reason string explaining its IMV Action Recommendation and IMV Evaluation Result.

With the Java Platform Binding for IF-IMV, attribute values for the Preferred Language attribute are represented as a String which explains the reason for the IMV's IMV Action Recommendation and IMV Evaluation Result. No standard format for the reason string has been defined. Any format is permissible and MUST be accommodated by the TNCS. The string is not terminated by a NUL character since this convention is not employed in the Java programming language. Thus, the Java Platform Binding for IF-IMV uses essentially the same attribute value as the abstract binding but translates it into the types and classes native to the Java platform.

4.3.9.3 Reason Language Attribute

The Reason Language attribute allows an IMV to indicate to the TNCS which language or languages were used in a reason string.

With the Java Platform Binding for IF-IMV, attribute values for the Reason Language attribute are represented as a String containing an RFC 3066 language tag. The string is not terminated by a NUL character since this convention is not employed in the Java programming language. Thus, the Java Platform Binding for IF-IMV uses essentially the same attribute value as the abstract binding but translates it into the types and classes native to the Java platform.

4.3.9.4 Maximum Round Trips Attribute

The Maximum Round Trips attribute allows an IMVConnection to indicate to the IMCs the maximum number of round trips the underlying transport supports.

With the Java Platform Binding for IF-IMV, attribute values for the Maximum Round Trips attribute are represented as a java.lang.Integer representing the maximum number of round trips allowed by the underlying transport. A value of zero (0) indicates that the maximum number of round trips for this connection is unknown, and a value of java.lang.Integer.MAX_VALUE indicates that the number of round trips is unlimited.

4.3.9.5 Maximum Message Size Attribute

The Maximum Message Size attribute allows an IMVConnection to indicate to the IMCs the maximum message size the underlying transport supports.

With the Java Platform Binding for IF-IMV, attribute values for the Maximum Message Size attribute are represented as a java.lang.Integer representing the maximum number of round trips allowed by the underlying transport. A value of zero (0) indicates that the maximum message size for this connection is unknown, and a value of java.lang.Integer.MAX_VALUE indicates that the maximum message size is unlimited.

4.3.9.6 Diffie-Hellman Pre-Negotiation Value (DHPN) Attribute

The DHPN attribute allows a IMVConnection that supports Diffie-Hellman Pre-Negotiation (as described in IF-T for Tunneled EAP Methods) to provide to a PTS-IMV or other IMVs a Unique-Value-1, so that this value can be used for verifying the PTS-IMC's Integrity Report.

With the Java Platform Binding for IF-IMV, attribute values for the DHPN attribute MUST be represented as a byte[20] array representing the Unique-Value-1.

4.3.9.7 Has Long Types Attribute

The Has Long Types attribute allows a TNCS to indicate to the IMVs that a particular connection supports long message types.

With the Java Platform Binding for IF-IMV, attribute values for the Has Long Types attribute are represented as a java.lang.Boolean indicating if the connection supports long message types. A value of java.lang.Boolean.TRUE indicates that this connection supports long message types, and a value of java.lang.Boolean.FALSE indicates that the connection does not support long message types.

4.3.9.8 Has Exclusive Attribute

The Has Exclusive attribute allows a TNCS to indicate to the IMVs that a particular connection supports exclusive delivery of messages.

With the Java Platform Binding for IF-IMV, attribute values for the Has Exclusive attribute are represented as a java.lang.Boolean indicating if the connection supports exclusive delivery. A value of java.lang.Boolean.TRUE indicates that this connection supports exclusive delivery, and a value of java.lang.Boolean.FALSE indicates that the connection does not support exclusive delivery.

4.3.9.9 Has SOH Attribute

The Has SOH attribute allows a TNCS to indicate to the IMVs that a particular connection supports SOH functions like `TNC_TNCS_SendMessageSOH`.

With the Java Platform Binding for IF-IMV, attribute values for the Has SOH attribute are represented as a java.lang.Boolean indicating if the connection supports SOH functions. A value of java.lang.Boolean.TRUE indicates that this connection supports SOH functions, and a value of java.lang.Boolean.FALSE indicates that the connection does not support SOH functions.

4.3.9.10 SOH Attribute

The SOH attribute allows IMVs to obtain a copy of the full SOH that was received from the TNCS on a particular connection.

With the Java Platform Binding for IF-IMV, attribute values for the SOH attribute are represented as byte[] array. Thus, the Java Platform Binding for IF-IMV uses essentially the same attribute value as the abstract binding but translates it into the types and classes native to the Java platform.

4.3.9.11 SSOH Attribute

The SSOH attribute allows IMVs to obtain a copy of the SSOH that was received from the TNCS on a particular connection.

With the Java Platform Binding for IF-IMV, attribute values for the SSOH attribute are represented as byte[] array. Thus, the Java Platform Binding for IF-IMV uses essentially the same attribute value as the abstract binding but translates it into the types and classes native to the Java platform.

4.3.9.12 IF-TNCCS Protocol Attribute

The IF-TNCCS Protocol attribute allows IMVs to determine which IF-TNCCS Protocol or similar protocol is being used for a particular connection.

With the Java Platform Binding for IF-IMV, attribute values for the IF-TNCCS Protocol attribute are represented as a String. The string is not terminated by a NUL character, since this convention is not employed in the Java programming language. Thus, the Java Platform Binding for IF-IMV uses essentially the same attribute value as the abstract binding but translates it into the types and classes native to the Java platform.

4.3.9.13 IF-TNCCS Version Attribute

The IF-TNCCS Version attribute allows IMVs to determine the version of the IF-TNCCS Protocol or similar protocol that is being used for a particular connection.

With the Java Platform Binding for IF-IMV, attribute values for the IF-TNCCS Version attribute are represented as a String. The string is not terminated by a NUL character, since this convention is not employed in the Java programming language. Thus, the Java Platform Binding for IF-IMV uses essentially the same attribute value as the abstract binding but translates it into the types and classes native to the Java platform.

4.3.9.14 IF-T Protocol Attribute

The IF-T Protocol attribute allows IMVs to determine which IF-T Protocol or other transport protocol is being used for a particular connection.

With the Java Platform Binding for IF-IMV, attribute values for the IF-T Protocol attribute are represented as a String. The string is not terminated by a NUL character, since this convention is not employed in the Java programming language. Thus, the Java Platform Binding for IF-IMV uses essentially the same attribute value as the abstract binding but translates it into the types and classes native to the Java platform.

4.3.9.15 IF-T Version Attribute

The IF-T Version attribute allows IMVs to determine the version of the IF-T Protocol or other transport protocol that is being used for a particular connection.

With the Java Platform Binding for IF-IMV, attribute values for the IF-T Version attribute are represented as a String. The string is not terminated by a NUL character, since this convention is not employed in the Java programming language. Thus, the Java Platform Binding for IF-IMV uses essentially the same attribute value as the abstract binding but translates it into the types and classes native to the Java platform.

4.3.9.16 Primary IMV ID Attribute

The Primary IMV ID attribute indicates the unique identifier of the IMV assigned by the TNCS when the TNCS loaded this IMV.

With the Java Platform Binding for IF-IMV, attribute values for the Primary IMV ID attribute are represented as a java.lang.Long.

4.3.9.17 TLS-Unique Attribute

The TLS-Unique attribute allows an IMVConnection that supports the TLS-Unique value (as described in PT-TLS) to provide to a PTS-IMV or other IMVs a TLS-Unique value, so that this value can be used in the TPM_Quote operation.

With the Java Platform Binding for IF-IMV, attribute values for the TLS-Unique attribute MUST be represented as a byte array representing the TLS-Unique value.

4.3.10 Platform-Specific Bindings for Basic Types

With the Java Platform Binding, the basic data types defined in the IF-IMV abstract API are mapped as follows:

The `TNC_UInt32` type is mapped to Java's `long` type.

The `TNC_BufferReference` type is mapped to a byte array (`byte []`). The value `NULL` is allowed for a `TNC_BufferReference` only where explicitly permitted in this specification.

Since Java does not have an equivalent of C's `typedef`, the Java types are used in the Java interface definitions.

4.3.11 Platform-Specific Bindings for Derived Types

With the Java Platform Binding, the platform-specific derived data types defined in the IF-IMV abstract API are mapped as follows:

The `TNC_MessageTypeList`, `TNC_VendorIDList`, and `TNC_MessageSubtypeList` types are mapped to an array of longs (`long []`). The value `NULL` is allowed for these types only where explicitly permitted in this specification.

4.3.12 Interface and Class Definitions

The IF-IMV specification defines a set of interfaces that must be implemented by TNC Servers and IMVs that implement the Java Platform Binding for IF-IMV. These interfaces allow the TNC Servers and IMVs to interact with each other in a well-defined and predictable manner. Figure 1 illustrates the primary interfaces in the Java Platform Binding for IF-IMV, as they might be implemented by a typical TNC Server and IMV.

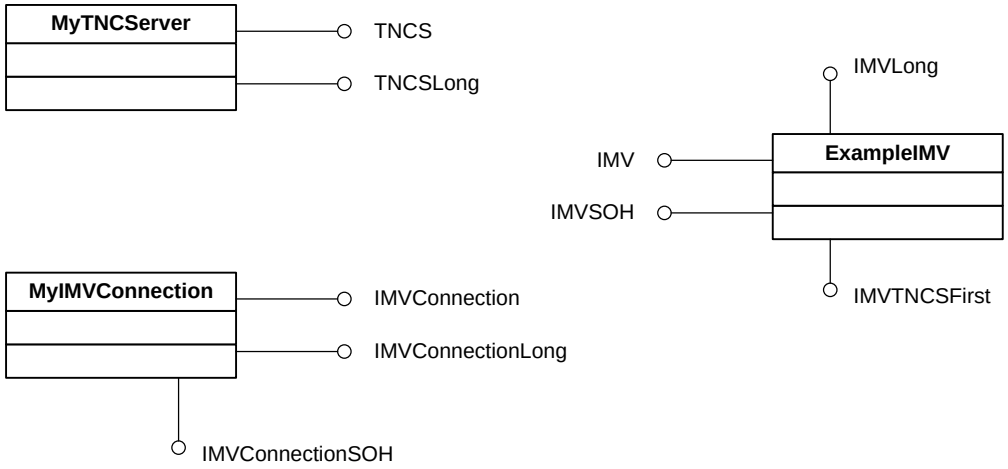


Figure 1: Typical Interface and Class Relationships

In this example, MyTNCServer implements the `TNCS` and `TNCSLong` interfaces, because it not only supports the basic methods required of all TNC Servers but also supports IF-TNCCS 2.0 and therefore supports the extra methods defined in `TNCSLong`. The implementer of MyTNCServer has also implemented a separate class named MyIMVConnection, which implements `IMVConnection` (including the basic methods used by an IMV to send messages and perform other operations on a connection) and also `IMVConnectionLong` and `IMVConnectionSOH`, because this class supports the extra methods included in those interfaces. Finally, ExampleIMV was created by another implementer. It supports not only the basic `IMV` interface but also several other optional interfaces, because its implementer enthusiastically decided to implement support for lots of special features like long message types and having the TNCS send the first messages in a handshake.

When MyTNCServer examined the `tnc_config` file, it used the fully qualified class name and JAR file path in that file to create an instance of ExampleIMV. Once this has been done, MyTNCServer can discover which interfaces are supported by this IMV by using the `instanceof` operator to check whether the ExampleIMV object, which must implement the `IMV` interface, also implements the `IMVLong` interface. If so, MyTNCServer can cast the ExampleIMV object to the `IMVLong` type and use the methods in that interface. The same mechanism is also used by ExampleIMV to discover that the `TNCSLong` interface is implemented by MyTNCServer. This extensibility mechanism will be used in future versions of IF-IMV, too. New interfaces will be defined in order to add new methods. New methods will not be added to existing interfaces, thus enabling new features to be detected dynamically using the `instanceof` operator.

Here are interface and class definitions for the Java Platform Binding for the IF-IMV API.

4.3.12.1 TNCException (TNCException.java)

org.trustedcomputinggroup.tnc

```
java.lang.Object
  └java.lang.Throwable
      └java.lang.Exception
          └org.trustedcomputinggroup.tnc.TNCException
```

All Implemented Interfaces:

java.io.Serializable

```
public class TNCException
extends java.lang.Exception
```

An exception that provides information on IF-IMC/IF-IMV errors. This exception class which wraps the result codes defined in the IF-IMC and IF-IMV Abstract API MUST be implemented by all TNCCs and TNCSs.

Each method in the IF-IMC/IF-IMV API throws an exception to indicate reason for failure. IMCs, IMVs, TNCCs and TNCSs MUST be prepared for any method to throw aTNCException.

This class defines a set of standard result codes. Vendor-specific result codes may be used but must be constructed as described in the abstract API. Any unknown result code SHOULD be treated as equivalent to TNC_RESULT_OTHER.

If an IMC or IMV method returns TNC_RESULT_FATAL, then the IMC or IMV has encountered a permanent error. The TNCC or TNCS SHOULD call the IMC or IMV's terminate method as soon as possible. The TNCC or TNCS MAY then try to reinitialize the IMC or IMV with the IMC or IMV's initialize method or try other measures.

If a TNCC or TNCS method returns TNC_RESULT_FATAL, then the TNCC or TNCS has encountered a permanent error.

Field Detail

`public static final long` **`TNC_RESULT_NOT_INITIALIZED`**
The IMC or IMV's `initialize` method has not been called.

`public static final long` **`TNC_RESULT_ALREADY_INITIALIZED`**
The IMC or IMV's `initialize` method was called twice without a call to the IMC or IMV's `terminate` method.

`public static final long` **`TNC_RESULT_CANT_RETRY`**
TNCC or TNCS cannot attempt handshake retry.

`public static final long` **`TNC_RESULT_WONT_RETRY`**
TNCC or TNCS refuses to attempt handshake retry.

`public static final long` **`TNC_RESULT_INVALID_PARAMETER`**
Method parameter is not valid.

`public static final long` **`TNC_RESULT_CANT_RESPOND`**
IMC or IMV cannot respond now.

`public static final long` **`TNC_RESULT_ILLEGAL_OPERATION`**
Illegal operation attempted.

`public static final long` **`TNC_RESULT_EXCEEDED_MAX_ROUND_TRIPS`**
Exceeded maximum round trips supported by the underlying protocol.

`public static final long` **`TNC_RESULT_EXCEEDED_MAX_MESSAGE_SIZE`**
Exceeded maximum message size supported by the underlying protocol.

`public static final long` **`TNC_RESULT_NO_LONG_MESSAGE_TYPES`**
Connection does not support long message types.

`public static final long` **`TNC_RESULT_NO_SOH_SUPPORT`**
Connection does not support SOH.

`public static final long` **`TNC_RESULT_OTHER`**
Unspecified error.

`public static final long` **`TNC_RESULT_FATAL`**
Unspecified fatal error.

Constructor Detail

`public` **`TNCException`**`()`
Constructs a `TNCException` object; the `resultCode` field defaults to TNC_RESULT_OTHER.

`public` **`TNCException`**`(java.lang.String s, long resultCode)`
Constructs a fully-specified `TNCException` object.
Parameters:
`s` - a description of the exception
`resultCode` - TNC result code

Method Detail

`public long` **`getResultCode`**`()`
Retrieves the TNC result code for this `TNCException` object.
Returns:
the TNC result code

4.3.12.2 TNCConstants (TNCConstants.java)

org.trustedcomputinggroup.tnc

`public interface` **`TNCConstants`**

A collection of well known or common constants to be used by the IMC and IMV packages.

Field Detail

`static final long` **`TNC_CONNECTION_STATE_CREATE`**
Network connection created.

`static final long` **`TNC_CONNECTION_STATE_HANDSHAKE`**
Handshake about to start.

`static final long` **`TNC_CONNECTION_STATE_ACCESS_ALLOWED`**
Handshake completed. TNC Server recommended that requested access be allowed.

`static final long` **`TNC_CONNECTION_STATE_ACCESS_ISOLATED`**
Handshake completed. TNC Server recommended that isolated access be allowed.

`static final long` **`TNC_CONNECTION_STATE_ACCESS_NONE`**
Handshake completed. TNCS Server recommended that no network access be allowed.

`static final long` **`TNC_CONNECTION_STATE_DELETE`**

About to delete network connection . Remove all associated state.

`static final long` **`TNC_VENDORID_ANY`**
Wild card matching any vendor ID.

`static final long` **`TNC_SUBTYPE_ANY`**
Wild card matching any message subtype.

`static final long` **`TNC_ATTRIBUTEID_PREFERRED_LANGUAGE`**
Preferred human-readable language(s) as an Accept-Language header (type String, may get from a TNCS or IMVConnection)

`static final long` **`TNC_ATTRIBUTEID_REASON_STRING`**
Reason for IMV Recommendation (type String, may set for an IMVConnection)

`static final long` **`TNC_ATTRIBUTEID_REASON_LANGUAGE`**
Language(s) for Reason String as an RFC 3066 language tag (type String, may set for an IMVConnection)

`static final long` **`TNC_ATTRIBUTEID_MAX_ROUND_TRIPS`**
Maximum round trips supported by the underlying protocol (type Integer, may get from an IMVConnection)

`static final long` **`TNC_ATTRIBUTEID_MAX_MESSAGE_SIZE`**
Maximum message size supported by the underlying protocol (type Integer, may get from an IMVConnection)

`static final long` **`TNC_ATTRIBUTEID_DHPN_VALUE`**
Diffie-Hellman Pre-Negotiation value provided by the underlying protocol (type byte[], may get from an IMVConnection)

`static final long` **`TNC_ATTRIBUTEID_HAS_LONG_TYPES`**
Flag indicating if the connection supports long message types (type Boolean, may get from an IMVConnection)

`static final long` **`TNC_ATTRIBUTEID_HAS_EXCLUSIVE`**
Flag indicating if the connection supports exclusive delivery of messages (type Boolean, may get from an IMVConnection)

`static final long` **`TNC_ATTRIBUTEID_HAS_SOH`**
Flag indicating if the connection supports SOH functions (type Boolean, may get from an IMVConnection)

`static final long` **`TNC_ATTRIBUTEID_SOH`**
Contents of SOH (type byte [], may get from an IMVConnection)

`static final long` **`TNC_ATTRIBUTEID_SSOH`**
Contents of SSOH (type byte [], may get from an IMVConnection)

`static final long` **`TNC_ATTRIBUTEID_IFTNCCS_PROTOCOL`**
IF-TNCCS Protocol Name (type String, may get from an IMVConnection)

`static final long` **`TNC_ATTRIBUTEID_IFTNCCS_VERSION`**
IF-TNCCS Protocol Version (type String, may get from an IMVConnection)

`static final long` **`TNC_ATTRIBUTEID_IFT_PROTOCOL`**

IF-T Protocol Name (type String, may get from an IMVConnection)

`static final long` **`TNC_ATTRIBUTEID_IFT_VERSION`**
IF-T Protocol Version (type String, may get from an IMVConnection)

`static final long` **`TNC_ATTRIBUTEID_PRIMARY_IMV_ID`**
Primary ID for IMV (type Long, may get from a TNCS or an IMVConnection)

`static final long` **`TNC_ATTRIBUTEID_TLS_UNIQUE`**
TLS-Unique value provided by the underlying protocol (type byte[], may get from a IMVConnection)

`static final long` **`TNC_ATTRIBUTEID_AR_IDENTITIES`**
Identities associated with AR (type byte[], may get from a IMVConnection)

`static final long` **`TNC_ID_UNKNOWN`**
Unknown address identifier type

`static final long` **`TNC_ID_IPV4_ADDR`**
IPv4 address identifier type

`static final long` **`TNC_ID_IPV6_ADDR`**
IPv6 address identifier type

`static final long` **`TNC_ID_FQDN`**
Fully-qualified domain name identifier type

`static final long` **`TNC_ID_EMAIL_ADDR`**
email address identifier type

`static final long` **`TNC_ID_USERNAME`**
username identifier type

`static final long` **`TNC_ID_X500_DN`**
X.500 Distinguished Name identifier type

`static final long` **`TNC_SUBJECT_UNKNOWN`**
Unknown subject type

`static final long` **`TNC_SUBJECT_MACHINE`**
Machine subject type

`static final long` **`TNC_SUBJECT_USER`**
User subject type

`static final long` **`TNC_AUTH_UNKNOWN`**
Unknown authentication type

`static final long` **`TNC_AUTH_X509_CERT`**
X.509 certificate authentication type

`static final long` **`TNC_AUTH_PASSWORD`**
Password authentication type

`static final long` **`TNC_AUTH_SIM`**
SIM authentication type

4.3.12.3 TNCS Interface (TNCS.java)

org.trustedcomputinggroup.tnc.ifimv

`public interface` **`TNCS`**

These methods are implemented by the TNC Server and called by the IMV.

Method Detail

```
void reportMessageTypes(IMV imv,
                        long[] supportedTypes)
                        throws TNCException
```

An IMV calls this method to inform a TNCS about the set of message types the IMV is able to receive. Often, the IMV will call this method from the IMV's initialize method.

A list of message types is contained in the supportedTypes parameter. The supportedTypes parameter may be null to represent no message types.

All TNC Servers MUST implement this method. The TNC Server MUST NOT ever modify the list of message types and MUST NOT access this list after the TNCS' reportMessageTypes method has returned. Generally, the TNC Server will copy the contents of this list before returning from this method. TNC Servers MUST support any message type.

Note that although all TNC Servers must implement this method, some IMVs may never call it if they don't support receiving any message types. This is acceptable. In such a case, the TNC Server MUST NOT deliver any messages to the IMV.

If an IMV requests a message type whose vendor ID is TNC_VENDORID_ANY and whose subtype is TNC_SUBTYPE_ANY it will receive all messages with any message type. This message type is 0xffffffff. If an IMV requests a message type whose vendor ID is NOT TNC_VENDORID_ANY and whose subtype is TNC_SUBTYPE_ANY, it will receive all messages with the specified vendor ID and any subtype. If an IMV calls the TNCS' reportMessageTypes method more than once, the message type list supplied in the latest call supplants the message type lists supplied in earlier calls.

Parameters:
`imv` - the IMV reporting its message types
`supportedTypes` - the message types the IMV wishes to receive
Throws:
`TNCException`

```
void requestHandshakeRetry(IMV imv,
                           long reason)
                           throws TNCException
```

IMVs can call this method to ask a TNCS to retry an Integrity Check Handshake for all current network connections. The IMV MUST pass itself as the imv parameter and one of the handshake retry reasons listed in IMVConnection as the reason parameter.

TNCSs MAY check the parameters to make sure they are valid and throw an exception if not, but TNCSs are not required to make these checks. The reason parameter explains why the IMV is requesting a handshake retry. The TNCS MAY use this in deciding whether to attempt the handshake retry. As noted in the Abstract API, TNCSs are not required to honor IMV requests for handshake retry (especially since handshake retry may not be possible or may interrupt network connectivity). An IMV MAY call this method at any time, even if an Integrity Check Handshake is currently underway. This is useful if the IMV suddenly gets important information but has already finished its dialog with the IMC, for instance. As always, the TNCS is not required to honor the request for handshake retry.

If the TNCS cannot attempt the handshake retry, it SHOULD throw a TNCException with result code TNC_RESULT_CANT_RETRY. If the TNCS could attempt to retry the handshake but chooses not to, it SHOULD throw the TNC_RESULT_WONT_RETRY exception. If the TNCS intends to retry the handshake, it SHOULD throw a TNCException with result code TNC_RESULT_WONT_RETRY. The IMV MAY use this information in displaying diagnostic and progress messages.

Parameters:
`imv` - IMV object
`reason` - reason for retry handshake request
Throws:
`TNCException`

```
java.lang.Object getAttribute(long attributeID)
                              throws TNCException
```

An IMV calls this method to get the value of the attribute identified by `attributeID` for this TNCS.

This function is optional. The TNCS is not required to implement it. If it is not implemented for this TNCS, it MUST throw an UnsupportedOperationException. IMVs MUST work properly if a TNCS does not implement this function.

The IMV MUST pass a standard or vendor-specific attribute ID as the attributeID parameter. If the TNCS does not recognize the attribute ID, it SHOULD throw a TNCException with the TNC_RESULT_INVALID_PARAMETER result code. If the TNCS recognizes the attribute ID but does not have an attribute value for the requested attribute ID for this TNCS, it SHOULD also throw a TNCException with the TNC_RESULT_INVALID_PARAMETER result code.

The return value is an Object that represents the attribute value requested. The IMV must cast this Object to the class documented in the description of that specific attribute to get the desired value. All Objects returned by this method SHOULD be immutable.

Parameters:
`attributeID` - the attribute ID of the desired attribute
Returns:
the attribute value
Throws:
`TNCException`

```
void setAttribute(long attributeID,
                  java.lang.Object attributeValue)
                  throws TNCException
```

An IMV calls this method to set the value of the attribute identified by `attributeID` for this TNCS.

This function is optional. The TNCS is not required to implement it. If it is not implemented for this TNCS, it MUST throw an `UnsupportedOperationException`. IMVs MUST work properly if a TNCS does not implement this function.

The IMV MUST pass a standard or vendor-specific attribute ID as the attributeID parameter. If the TNCS does not recognize the attribute ID, it SHOULD throw a `TNCException` with the `TNC_RESULT_INVALID_PARAMETER` result code. If the TNCS recognizes the attribute ID but does not support setting an attribute value for the

requested attribute ID for this TNCS, it SHOULD also throw a `TNCException` with the `TNC_RESULT_INVALID_PARAMETER` result code.

For the attributeValue parameter, the IMV MUST pass an `Object` that represents the new attribute value (or `null` if permitted for the specified attribute). This `Object` must actually be an instance of the class documented in the description of the specified attribute. The `Object` SHOULD be immutable. If the TNCS has any uncertainty about it SHOULD copy the object. The TNCS MAY check the `Object` and throw a `TNCException` if it is not a valid value for the specified attribute.

Parameters:
`attributeID` - the attribute ID of the attribute to be set
`attributeValue` - the new value to be set for this attribute
Throws:
`TNCException`

4.3.12.4 TNCSLong Interface (TNCSLong.java)

org.trustedcomputinggroup.tnc.ifimv

Interface TNCSLong

`public interface` **`TNCSLong`**
This interface can be implemented by an object that already supports the TNCS interface to indicate that this object is a TNCS that supports reportMessageTypesLong. To check whether an object implements this interface, use the `instanceof` operator. If so, cast the object to the interface type and use the methods in this interface.

Method Detail

```
void reportMessageTypesLong(IMV imv,
                            long[] supportedVendorIDs,
                            long[] supportedSubtypes)
                            throws TNCException
```

A call to this method is used to inform a TNCS about the set of message types that the IMV wishes to receive. This function supports long message types, unlike reportMessageTypes.

Often, the IMV will call this method from the IMV's initialize method. A list of Vendor IDs is contained in the supportedVendorIDs parameter, and a list of message subtypes is contained in the supportedSubtypes parameter. Both lists must have exactly the same number of entries. Either parameter may be null to represent no message types, which is equivalent to a zero length list. The values in the `supportedVendorIDs` list and the `supportedSubtypes` list are matched pairs that represent the (Vendor ID, Message Subtype) pairs that the IMV is able to receive.

This method is optional and is not supported by all TNCSs. IMVs can easily determine which TNCSs support this method by checking if they implement the TNCSLong interface. IMVs should recognize that many TNCSs do not support long types and

therefore will not support this method. In those cases, IMVs should be prepared to use the reportMessageTypes function. Simple IMVs that do not wish to send messages with long subtypes need not implement this function.

The TNC Server MUST NOT ever modify the list of message types and MUST NOT access this list after the reportMessageTypesLong() method has returned. Generally, the TNC Server will copy the contents of this list before returning from this method. TNC Servers MUST support any message type.

If an IMV requests a message type whose vendor ID is TNC_VENDORID_ANY and whose subtype is TNC_SUBTYPE_ANY, it will receive all messages with any message type. If an IMC requests a message type whose vendor ID is not TNC_VENDORID_ANY and whose subtype is TNC_SUBTYPE_ANY, it will receive all messages with the specified vendor ID and any subtype. If an IMV calls the TNCS's reportMessageTypes or reportMessageTypesLong method more than once, the message type list supplied in the latest call supplants the message type lists supplied in earlier calls.

Parameters:
`imv` - the IMV reporting its message types
`supportedVendorIDs` - the list of Vendor IDs the IMV wishes to receive
`supportedSubtypes` - the list of message subtypes the IMV wishes to receive
Throws:
`TNCException` - if a TNC error occurs

`long` **`reserveAdditionalIMVID`**`(IMV imv)`
`throws TNCException`

An IMV calls this method to reserve an additional IMV ID for itself. This method is optional. The TNCS is not required to implement it, but if it implements the TNCSLong interface, then it MUST implement this method. Since this method was not included in IF-IMV 1.2, many TNCSs do not support it. IMVs MUST work properly if a TNCS does not implement this method.

When the IF-TNCCS 2.0 [14] (and PB-TNC [21]) protocols are carrying an IF-M message, the IF-TNCCS protocol includes a header (TNCCS-IF-M-Message) housing several fields important to the processing of a received IF-M message. The IF-M Vendor ID and IF-M Subtype described in the IF-TNCCS specification are used by the TNCC and TNCS to route messages to IMC and IMV that have registered an interest in receiving messages for a particular type of component. Also present in the TNCCS-IF-M-Message header is a pair of fields that identify the IMC and IMV involved in the message exchange. The IMC and IMV Identifier fields are used for performing exclusive delivery of messages and as an indicator for correlation of received attributes. See the IF-TNCCS 2.0 protocol specification for more information on these fields.

Correlation of attributes is necessary when an IMC sends attributes describing multiple different implementations of a single type of component during an assessment, so the recipient IMV(s) need to be able to determine which attributes are describing the same implementation.

For example, a single IMC might report attributes about two installed VPN implementations on the endpoint. Because the individual attributes (except the Product Information attribute) do not include an indication of which VPN product they are describing, the recipient IMV needs something to perform this correlation.Therefore, for this example, the single VPN IMC would need to obtain two IMC Identifiers from the TNC Client and consistently use one with each of the VPN implementations reported during an

assessment. The VPN IMC would group all the attributes associated with a particular VPN implementation into a single IF-M message and send the message using the IMC Identifier it designates as going with the particular implementation. This approach allows the recipient IMV to recognize when attributes in future assessment messages also describe the same VPN implementation and to direct follow-up messages to the right IMC. Similarly, a single IMV may need to have multiple IMV IDs so that an IMC can send follow-up messages to the right IMV.

An IMV can call this function as many times it wishes. The TNCS SHOULD return a unique value every time. However, the TNCS may have a maximum number of additional IMV IDs that it supports. In that case the TNCS SHOULD return a `TNC_RESULT_OTHER` error if an IMV attempts to exceed this maximum. The TNCS is not required to reserve IMV IDs in a specific order.

Parameters:
`imv` - the IMV requesting additional IMV IDs
Throws:
`TNCException` - if a TNC error occurs

4.3.12.5 IMV Interface (IMV.java)

org.trustedcomputinggroup.tnc.ifimv

Interface IMV

```
public interface IMV
```

An Integrity Measurement Verifier (IMV). These methods are implemented by the IMV and called by the TNC Server.

Method Detail

```
void initialize(TNCS tncs)
                throws TNCException
```

Initializes the IMV. All IMVs MUST implement this Method. The TNC Server supplies itself as a parameter so the IMV can call the TNCS as needed.

The TNC Server MUST NOT call any other IF-IMV API Methods for an IMV until it has successfully completed a call to the IMV's initialize method. Once a call to this method has completed successfully, this method MUST NOT be called again for a particular IMV-TNCS pair until a call to the IMV's terminate method has completed successfully.

Parameters:
`tncs` - the TNC Server
Throws:
`TNCException` - if a TNC error occurs

```
void terminate()
                throws TNCException
```

Closes down the IMV. The TNC Server calls this method when all work is complete and the TNCS is preparing to shut down or when the IMV throws a TNC_RESULT_FATAL exception. Once a call to an IMV's terminate method is made, the TNC Server MUST

NOT call the IMV except to call the IMV's initialize method (which may not succeed if the IMV cannot reinitialize itself). Even if the IMV returns an error from this method, the TNC Server MAY continue with its unload or shutdown procedure.

Throws:

TNCException - if a TNC error occurs

```
void notifyConnectionChange(IMVConnection c,
                            long newState)
                            throws TNCException
```

Informs the IMV that the state of IMVConnection c has changed to newState. The TNCConstants interface lists all the possible values of the new state for this version of the IF-IMV API. The TNCS MUST NOT use any other values with this version of IF-IMV.

IMVs that want to track the state of network connections or maintain per-connection data structures SHOULD implement this method. Other IMVs MAY implement it. If an IMV chooses to not implement this method it MUST throw an UnsupportedOperationException.

If the state is TNC_CONNECTION_STATE_CREATE, the IMV SHOULD note the creation of a new network connection.

If the state is TNC_CONNECTION_STATE_HANDSHAKE, an Integrity Check Handshake is about to begin.

If the state is TNC_CONNECTION_STATE_DELETE, the IMV SHOULD discard any state pertaining to this network connection and MUST NOT pass this network connection to the TNC Server after this method returns.

Parameters:

c - the IMV Connection

newState - new network connection state

Throws:

TNCException

```
void receiveMessage(IMVConnection c,
                    long messageType,
                    byte[] message)
                    throws TNCException
```

The TNC Server calls this method to deliver a message to the IMV. The message is contained in the buffer referenced by message. The type of the message is indicated by messageType. The message must be from an IMC (or a TNCC or other party acting as an IMC).

If IF-TNCCS-SOH is used for a connection, and the IMV and TNCS implement receiveMessageSOH(), the TNCS SHOULD use that method to deliver the contents of SOHReportEntries instead of using receiveMessage. However, if IF-TNCCS-SOH is used for a connection, but either the IMV or the TNCS does not implement receiveMessageSOH, receiveMessage SHOULD be used instead. For each SOHReportEntry, the value contained in the first System-Health-ID attribute should be compared to the messageType values previously supplied in the IMV's most recent call to reportMessageTypes or reportMessageTypesLong. The SOHReportEntry should be delivered to each IMV that has a match. If an IMV that does not support receiveMessageSOH (or when the TNCS does not support that method), receiveMessage SHOULD be employed. In that case, the TNCS MUST pass, in the message parameter, a reference to a buffer containing the Data field of the first Vendor-

Specific attribute whose Vendor ID matches the value contained in the System-Health-ID. If no such Vendor-Specific attribute exists, the SOHReportEntry MUST NOT be delivered to this IMV. The TNCS MUST pass, in the messageType parameter, the value contained in the first System-Health-ID attribute.

The IMV SHOULD send any IMC-IMV messages it wants to send as soon as possible after this method is called and then return from this method to indicate that it is finished sending messages in response to this message.

As with all IMV methods, the IMV SHOULD NOT wait a long time (definitely not more than a second) before returning from the IMV receiveMessage() method. To do otherwise would risk delaying the handshake indefinitely. A long delay might frustrate users or exceed network timeouts (PDP, PEP or otherwise).

The IMV should implement this method if it wants to receive messages. Most IMVs will do so, since they will base their IMV Action Recommendations on measurements received from the IMC. However, some IMVs may base their IMV Action Recommendations on other data such as reports from intrusion detection systems or scanners. Those IMVs need not implement this method.

The IMV MUST NOT ever modify the buffer contents and MUST NOT access the buffer after the IMV receiveMessage() method has returned. If the IMV wants to retain the message, it should copy it before returning from this method.

The message parameter may be null to represent an empty message. In the messageType parameter, the TNCS MUST pass the type of the message. This value MUST match one of the TNC_MessageType values previously supplied by the IMV to the TNCS in the IMV's most recent call to the TNCS reportMessageTypes method. IMVs MAY check these parameters to make sure they are valid and return an error if not, but IMVs are not required to make these checks.

Parameters:
c - the IMVConnection
`messageType` - the message type that is being delivered to the IMV
`message` - the message that is being delivered to the IMV
Throws:
`TNCException`

`void` **`solicitRecommendation`**`(IMVConnection c)`
`throws TNCException`

The TNC Server calls this method at the end of an Integrity Check Handshake (after all IMC-IMV messages have been delivered) to solicit recommendations from IMVs that have not yet provided a recommendation. The TNCS MUST NOT call this method for an IMV for a particular connection if that IMV has already called `provideRecommendation` on that connection since the TNCS last called `notifyConnectionChange` for that IMV with that connection. If an IMV is not able to provide a recommendation at this time, it SHOULD call the TNCS provideRecommendation() method with the recommendation parameter set to TNC_IMV_ACTION_RECOMMENDATION_NO_RECOMMENDATION. If an IMV returns from this method without calling the TNCS' provideRecommendation method, the TNCS MAY consider the IMV's Action Recommendation to be TNC_IMV_ACTION_RECOMMENDATION_NO_RECOMMENDATION. The TNCS MAY take other actions, such as logging this IMV behavior, which is erroneous.

All IMVs MUST implement this method.

Note that a TNCC or TNCS MAY cut off IMC-IMV communications at any time for any reason, including limited support for long conversations in underlying protocols, user or administrator intervention, or policy. If this happens, the TNCS will return TNC_RESULT_ILLEGAL_OPERATION from sendMessage and call the solicitRecommendation method to elicit IMV Action Recommendations based on the data they have gathered so far.

Parameters:
c - IMV connection
Throws:
TNCException

`void` **`batchEnding`**`(IMVConnection c)`
`throws TNCException`

The TNC Server calls this method to notify IMVs that all IMC messages received in a batch have been delivered and this is the IMV's last chance to send a message in the batch of IMV messages currently being collected. An IMV MAY implement this method if it wants to perform some actions after all the IMC messages received during a batch have been delivered (using the IMV's receiveMessage(), receiveMessageLong(), or receiveMessageSOH() methods). For instance, if an IMV has not received any messages from an IMC it may conclude that its IMC is not installed on the endpoint and may decide to call the TNCS' provideRecommendation method with the recommendation parameter set to TNC_IMV_ACTION_RECOMMENDATION_NO_ACCESS.

An IMV MAY call the TNCS' sendMessage(), sendMessageSOH(), or sendMessageLong() methods from this method. As with all IMV methods, the IMV SHOULD NOT wait a long time (definitely not more than a second) before returning from the batchEnding method. To do otherwise would risk delaying the handshake indefinitely. A long delay might frustrate users or exceed network timeouts (PDP, PEP or otherwise).

Parameters:
c - IMV connection
Throws:
TNCException

4.3.12.6 IMVTNCSFirst Interface (IMVTNCSFirst.java)

org.trustedcomputinggroup.tnc.ifimv

Interface IMVTNCSFirst

`public interface` **`IMVTNCSFirst`**

This interface can be implemented by an object that already supports the IMV interface to indicate that this object is an IMV that supports `beginHandshake`. To check whether an object implements this interface, use the `instanceof` operator. If so, cast the object to the interface type and use the methods in this interface.

Method Detail

`void` **`beginHandshake`**`(IMVConnection c)`

throws TNCException

The TNC Server calls this method to indicate that an Integrity Check Handshake is beginning and solicits messages from IMVs for the first batch. The IMV SHOULD send any IMC-IMV messages it wants to send, as soon as possible after this method is called, and then return from this method to indicate that it is finished sending messages for this batch.

IMVs are not required to implement, this method but any IMV that implements the IMVTNCSFirst interface MUST implement this method. An IMV should implement this method if it supports having the TNCS send the first batch of IMC-IMV messages in the TNC handshake. This is useful when an IMV must send instructions to the IMC before the IMC can send useful integrity messages. Simple IMVs that do not wish to send an initial message to the IMC need not implement this method.

Since this method was not included in IF-IMV 1.0 through 1.2, many IMVs do not implement it. TNCSs MUST work properly if an IMV does not implement this method. Likewise, many TNCSs do not implement this method, so IMVs MUST work properly if a TNCS does not implement this method. If an IMV implements this method, the TNCS SHOULD call this method if the TNCS is preparing to send the first batch of messages in a TNC handshake. Even if the TNCS and IMV both support this method and have used it for previous handshakes, a particular handshake may not involve having the TNCS send the first batch of messages. In that case, this method will not be called for that handshake. The TNCS and IMV MUST properly handle the situation where this method is supported by both the TNCS and IMV but is not called for a particular handshake because the TNCS will not be sending the first batch of messages.

As with all IMV methods, the IMV SHOULD NOT wait a long time (definitely not more than a second) before returning from `beginHandshake`. To do otherwise would risk delaying the handshake indefinitely. A long delay might frustrate users or exceed network timeouts (PDP, PEP or otherwise).

Parameters:
c - the IMVConnection object
Throws:
TNCException - if a TNC error occurs

4.3.12.7 IMVSOH Interface (IMVSOH.java)

org.trustedcomputinggroup.tnc.ifimv

Interface IMVSOH

`public interface` **`IMVSOH`**
This interface can be implemented by an object that already supports the IMV interface to indicate that this object is an IMV that supports functions specific to IF-TNCCS-SOH. To check whether an object implements this interface, use the `instanceof` operator. If so, cast the object to the interface type and use the methods in this interface.

Method Detail

`void` **`receiveMessageSOH`**`(IMVConnection c,`

```
                long systemHealthID,
                byte[] sohReportEntry)
                throws TNCException
```

The TNC Server calls this method (if supported by the TNCS and implemented by an IMV) to deliver an SOHReportEntry message to the IMV. This allows an IMV to receive the entire SOHReportEntry instead of just the contents of the Vendor-Specific attribute, as would be the case if the receiveMessage method was used.

A TNCS that supports IF-TNCCS-SOH SHOULD support this method, and a TNCS that supports this function SHOULD call this method for a particular IMV and connection, instead of calling receiveMessage, if the IMV implements this method and the connection uses IF-TNCCS-SOH. A TNCS MUST NOT call this method for a particular IMV and connection if either the IMV does not implement this function, or the connection does not use IF-TNCCS-SOH. A TNCS will often find that some IMVs loaded by that TNCS implement this function and some do not. In that case, when the TNCS is handling a connection that uses IF-TNCCS-SOH, the TNCS would call receiveMessageSOH for the IMVs that implement this method and receiveMessage for the IMVs that do not implement receiveMessageSOH. A TNCS MUST NOT call both receiveMessage and receiveMessageSOH for a single SOHReportEntry for a single IMV.

IMVs are not required to implement this method, but any IMV that implements the IMVSOH interface MUST implement this method. An IMV should implement receiveMessageSOH if it wants to receive the entire SOHReportEntry instead of just the Vendor-Specific attribute when IF-TNCCS-SOH is used. However, IMVs should recognize that many TNCSs do not support IF-TNCCS-SOH, and some TNCSs that do support IF-TNCCS-SOH will not support this function. Therefore, IMVs should be prepared to receive messages via receiveMessage in some cases when IF-TNCC-SOH is used. Simple IMVs that do not wish to receive the entire SOHReportEntry need not implement this function. IMV implementers may find that implementing receiveMessageSOH is more complex than implementing receiveMessage, since receiveMessageSOH must parse the SOHReportEntry, while receiveMessage only needs to parse the contents of the Vendor-Specific attribute within the SOHReportEntry. For many IMVs, the content of the Vendor-Specific attribute is all that they need. Therefore, IMVs are never required to implement receiveMessageSOH.

The content of the SOHReportEntry is contained in the buffer referenced by sohReportEntry. This parameter may be null to represent an empty message. The content of the first System-Health-ID attribute in the SOHReportEntry is indicated by systemHealthID. This value MUST match one of the message type values previously supplied by the IMV to the TNCS in the IMV's most recent call to the TNCS's reportMessageTypes() or reportMessageTypesLong() methods. IMVs MAY check these parameters to make sure they are valid and throw an exception if not, but IMVs are not required to make these checks.

The IMV SHOULD send any IMC-IMV messages it wants to send, as soon as possible after this method is called, and then return from this method to indicate that it is finished sending messages in response to this message.

As with all IMV methods, the IMV SHOULD NOT wait a long time (definitely not more than a second) before returning from receiveMessageSOH. To do otherwise would risk delaying the handshake indefinitely. A long delay might frustrate users or exceed network timeouts (PDP, PEP or otherwise).
The IMV MUST NOT ever modify the buffer contents and MUST NOT access the buffer after receiveMessageSOH has returned. If the IMV wants to retain the content of the SOHReportEntry, the IMV should copy it before returning from receiveMessageSOH.

Parameters:
c - the IMV connection object
`systemHealthID` - the type of message to be delivered
`sohReportEntry` - the message to be delivered
Throws:
`TNCException` - if a TNC error occurs

4.3.12.8 IMVLong Interface (IMVLong.java)

org.trustedcomputinggroup.tnc.ifimv

Interface IMVLong

`public interface` **`IMVLong`**
This interface can be implemented by an object that already supports the IMV interface to indicate that this object is an IMV that supports receiveMessageLong. To check whether an object implements this interface, use the `instanceof` operator. If so, cast the object to the interface type and use the methods in this interface.

Method Detail

```
void receiveMessageLong(IMVConnectionLong c,
                        long messageFlags,
                        long messageVendorID,
                        long messageSubtype,
                        byte[] message,
                        long sourceIMCID,
                        long destinationIMVID)
                        throws TNCException
```

The TNC Server calls this method to deliver a message to the IMV. This method provides several features that `receiveMessage` does not: longer (32 bit) vendor ID and message subtype fields, a `messageFlags` field, and the ability to specify a source IMC ID and destination IMV ID.

The message is contained in the buffer referenced by `message`. This parameter may be null to represent an empty message. The type of the message is indicated by the `messageVendorID` and `messageSubtype` parameters. The message must be from an IMC (or a TNCC or other party acting as an IMC). Any flags associated with the message are included in the `messageFlags` parameter. The `sourceIMCID` and `destinationIMVID` parameters indicate the IMC ID of the IMC that sent this message (if available) and either the IMV ID of the intended recipient (if the EXCL flag is set), or the IMV ID in response to whose message or messages this message was sent. If the EXCL flag is set, `destinationIMVID` MUST be either the primary IMV ID for this IMV (matching the value of the `TNC_AttributeID_PRIMARY_IMV_ID` attribute), or an additional IMV ID reserved when the IMV requested the TNCS to do so by calling `reserveAdditionalIMVID`. If the EXCL flag is not set, then `destinationIMVID` MAY be set to the wild card `TNC_IMVID_ANY`.

If an IF-TNCCS protocol that supports long types or exclusive delivery is used for a connection, and the IMV and TNCS implement `receiveMessageLong`, the TNCS SHOULD use this method to deliver messages instead of using `receiveMessage`. However, if the IMV does not implement `receiveMessageLong`, the TNCS SHOULD

use `receiveMessage` instead. Messages whose vendor ID or message subtype is too long to be represented in the parameters supported by `receiveMessage` MUST NOT be delivered to an IMV that does not support `receiveMessageLong`. This should be fine, since the IMV in question wouldn't have the ability to process such messages. Also, the IMV probably calls `reportMessageTypes` instead of `reportMessageTypesLong`, so it couldn't have expressed an interest in messages with long types except via wild cards. Messages with flags or source IMC IDs can be handled using the `receiveMessage` method.

The IMV SHOULD send any IMC-IMV messages it wants to send, as soon as possible after this method is called, and then return from this method to indicate that it is finished sending messages in response to this message.

As with all IMV methods, the IMV SHOULD NOT wait a long time (definitely not more than a second) before returning from `receiveMessageLong`. To do otherwise would risk delaying the handshake indefinitely. A long delay might frustrate users or exceed network timeouts (PDP, PEP or otherwise).

IMVs are not required to implement this method, but any IMV that implements the IMVLong interface MUST implement this method. An IMV should implement this method if it wants to receive messages with long types or flags or source IMC IDs. Simple IMVs need not implement this method. Since this method was not included in IF-IMV 1.2, many IMVs do not implement it. TNCSs MUST work properly if an IMV does not implement this method. Likewise, many TNCSs do not implement this method, so IMVs MUST work properly if a TNCS does not implement this method.

The IMV MUST NOT ever modify the buffer contents and MUST NOT access the buffer after `receiveMessageLong` has returned. If the IMV wants to retain the message, the IMV should copy it before returning from `receiveMessageLong`.

In the `messageVendorID` and `messageSubtype` parameters, the TNCS MUST pass the vendor ID and message subtype of the message. These values MUST match one of the values previously supplied by the IMV to the TNCS in the IMV's most recent call to `reportMessageTypes` or `reportMessageTypesLong`. The TNCS MUST NOT specify a message type whose vendor ID is 0xffffff or whose subtype is 0xff. These values are reserved for use as wild cards, as described in section 3.9.1. IMVs MAY check these parameters to make sure they are valid and throw an exception if not, but IMVs are not required to make these checks.

Any flags associated with the message are included in the `messageFlags` parameter. This may include the EXCL flag, but the TNCS MUST process that flag itself to ensure that a message with this flag set is only delivered to the intended recipient.

Parameters:
c - the IMVConnectionLong object
`messageFlags` – flags associated with the message
`messageVendorID` – vendor ID associated with the message
`messageSubtype` – message subtype associated with the message
`message` – the message to be delivered
`sourceIMCID` – source IMC ID for the message
`destinationIMVID` - destination IMV ID for message
Throws:
`TNCException` - if a TNC error occurs

4.3.12.9 IMVConnection Interface (IMVConnection.java)

org.trustedcomputinggroup.tnc.ifimv

Interface IMVConnection

`public interface` **`IMVConnection`**

The IMV and TNCS use this IMVConnection object to refer to the network connection when delivering messages and performing other operations relevant to the network connection. This helps ensure that IMV messages are sent to the right TNCC and IMCs, helps ensure that the IMV Action Recommendation is associated with the right endpoint, and helps the IMV match up messages from IMCs with any state the IMV may be maintaining from earlier parts of that IMC-IMV conversation (even extending across multiple Integrity Check Handshakes in a single network connection).

The TNCS MUST create a new IMVConnection object for each combination of an IMV and a connection. IMVConnection objects MUST NOT be shared between multiple IMVs.

Field Detail

`static final long` **`TNC_RETRY_REASON_IMV_IMPORTANT_POLICY_CHANGE`**
IMV policy has changed. It recommends handshake retry even if network connectivity must be interrupted;

`static final long` **`TNC_RETRY_REASON_IMV_MINOR_POLICY_CHANGE`**
IMV policy has changed. It requests handshake retry but not if network connectivity must be interrupted;

`static final long` **`TNC_RETRY_REASON_IMV_SERIOUS_EVENT`**
IMV has detected a serious event and recommends handshake retry even if network connectivity must be interrupted.

`static final long` **`TNC_RETRY_REASON_IMV_MINOR_EVENT`**
IMV has detected a minor event. It requests handshake retry but not if network connectivity must be interrupted.

`static final long` **`TNC_RETRY_REASON_IMV_PERIODIC`**
IMV wishes to conduct a periodic recheck. It recommends handshake retry but not if network connectivity must be interrupted.

`static final long` **`TNC_IMV_ACTION_RECOMMENDATION_ALLOW`**
IMV recommends allowing access.

`static final long` **`TNC_IMV_ACTION_RECOMMENDATION_NO_ACCESS`**
IMV recommends no access.

`static final long` **`TNC_IMV_ACTION_RECOMMENDATION_ISOLATE`**
IMV recommends limited access. This access may be expanded after remediation.

`static final long` **`TNC_IMV_ACTION_RECOMMENDATION_NO_RECOMMENDATION`**
IMV does not have a recommendation.

`static final long` **`TNC_IMV_EVALUATION_RESULT_COMPLIANT`**
AR complies with policy.

`static final long` **`TNC_IMV_EVALUATION_RESULT_NONCOMPLIANT_MINOR`**
AR is not compliant with policy. Non-compliance is minor.

`static final long` **`TNC_IMV_EVALUATION_RESULT_NONCOMPLIANT_MAJOR`**
AR is not compliant with policy. Non-compliance is major.

`static final long` **`TNC_IMV_EVALUATION_RESULT_ERROR`**
IMV is unable to determine policy compliance due to error.

`static final long` **`TNC_IMV_EVALUATION_RESULT_DONT_KNOW`**
IMV does not know whether AR complies with policy.

Method Detail

```
void sendMessage(long messageType,
                 byte[] message)
                 throws TNCException
```

Gives a message to the TNCS for delivery. The message is contained in the buffer referenced by the `message` parameter. The `message` parameter may be null which represent an empty message. The type of the message is indicated by the messageType parameter.

All IMVConnections MUST implement this method. An IMVConnection MUST NOT ever modify the buffer contents and MUST NOT access the buffer after the sendMessage method has returned. The IMVConnection will typically copy the message out of the buffer, queue it up for delivery, and return from this method.

The IMV MUST NOT call this method unless it has received a call to the IMV's receiveMessage method or the IMV's batchEnding method for this connection and the IMV has not yet returned from that method. If the IMV violates this prohibition, the TNCS SHOULD throw the TNC_RESULT_ILLEGAL_OPERATION exception. If an IMV really wants to communicate with an IMC at another time, it should call the IMVConnection's requestHandshakeRetry method.

Note that a TNCC or TNCS MAY cut off IMC-IMV communications at any time for any reason, including limited support for long conversations in underlying protocols, user or administrator intervention, or policy. If this happens, the IMVConnection's sendMessage method will throw a TNCException with result code TNC_RESULT_ILLEGAL_OPERATION and call the IMVs' solicitRecommendation() to elicit IMV Action Recommendations based on the data they have gathered so far. If the TNCS supports limiting the message size or number of round trips, the TNCS MUST throw a TNCException with result code TNC_RESULT_EXCEEDED_MAX_MESSAGE_SIZE or TNC_RESULT_EXCEEDED_MAX_ROUND_TRIPS respectively if the limits are exceeded. An IMV can get the Maximum Message Size and Maximum Number of Round Trips by using the related Attribute ID with the getAttribute method on the TNCS object. The IMV SHOULD adapt its behavior to accommodate these limitations if available.

The TNC Server MUST support any message type. However, the IMV MUST NOT specify a message type whose vendor ID is 0xffffff or whose subtype is 0xff. These values are reserved for use as wild cards, as described in the Abstract API. If the IMV violates this prohibition, the IMVConnection SHOULD throw a TNCException with result code TNC_RESULT_INVALID_PARAMETER. If the IMVConnection supports limiting the message size or number of round trips, the IMVConnection MUST return TNC_RESULT_EXCEEDED_MAX_MESSAGE_SIZE exception or TNC_RESULT_EXCEEDED_MAX_ROUND_TRIPS exception respectively if the limits

are exceeded. An IMV can get the Maximum Message Size and Maximum Number of Round Trips by using the related TNC_ATTRIBUTEID within the TNCS.getAttribute method. The IMV SHOULD adapt its behavior to accommodate these limitations if available.

Parameters:
`messageType` - the type of message to be delivered
`message` - the message to be delivered
Throws:
`TNCException` - if an error occurs

```
void requestHandshakeRetry(long reason)
                           throws TNCException
```

Asks a TNCS to retry an Integrity Check Handshake for this IMVConnection. The IMV MUST pass one of the handshake retry reasons listed in the Abstract API as the reason parameter.

TNCSs MAY check the parameters to make sure they are valid and throw an exception if not, but TNCSs are not required to make these checks. The reason parameter explains why the IMV is requesting a handshake retry. The TNCS MAY use this in deciding whether to attempt the handshake retry. As noted in the Abstract API, TNCSs are not required to honor IMV requests for handshake retry (especially since handshake retry may not be possible or may interrupt network connectivity). An IMV MAY call this method at any time, even if an Integrity Check Handshake is currently underway. This is useful if the IMV suddenly gets important information but has already finished its dialog with the IMC, for instance. As always, the TNCS is not required to honor the request for handshake retry.

If the TNCS cannot attempt the handshake retry, the IMVConnection SHOULD throw a TNCException with result code TNC_RESULT_CANT_RETRY. If the TNCS could attempt to retry the handshake but chooses not to, the IMVConnection SHOULD throw a TNCException with result code TNC_RESULT_WONT_RETRY. The IMV MAY use this information in displaying diagnostic and progress messages.

Parameters:
`reason` - handshake retry reason code
Throws:
`TNCException`

```
void provideRecommendation(long recommendation,
                           long evaluation)
                           throws TNCException
```

An IMV calls this method to deliver its IMV Action Recommendation and IMV Evaluation Result to the TNCS. The TNCS SHOULD use the recommendation value in determining its own TNCS Action Recommendation or decision about endpoint access. The TNC specifications do not specify how the TNCS does the recommendation value but it is certainly essential to have a recommendation from the IMV. The TNC specifications also do not specify what the TNCS does with the evaluation value. It may log it.

The IMV MUST pass one of the IMV Action Recommendation values listed in the Abstract API as the recommendation parameter and one of the IMV Evaluation Result values listed in the Abstract API as the evaluation parameter. TNCSs MAY check these values to make sure they are valid and throw an exception if not, but TNCSs are not required to make these checks.

The IMV should deliver its IMV Action Recommendation as soon as possible so that the TNCS can proceed with determining its own TNCS Action Recommendation. If the IMV receives a message from an IMC and is able to decide on an IMV Action Recommendation and deliver it to the TNCS before returning from the IMV receiveMessage method, it SHOULD do so. However, as always the IMV SHOULD return promptly to avoid a long delay that might frustrate users or exceed network timeouts (PDP, PEP or otherwise).

An IMV SHOULD NOT expect that it will be able to send IMC-IMV messages after calling the IMVConnection's provideRecommendation method. The TNCS may decide to terminate the handshake immediately based on the IMV Action Recommendation. For instance, IMVs SHOULD send remediation instructions before calling the IMVConnection's provideRecommendation method.

However, a TNCS MAY continue to deliver messages after an IMV calls the IMVConnection's provideRecommendation method, especially if other IMVs continue the dialog after the one IMV has rendered its decision. The IMV MUST be prepared for this. It MAY simply ignore these late messages or it MAY consider them and even change its recommendation by calling the IMVConnection's provideRecommendation method again. In this case, the TNCS SHOULD use the last recommendation received from an IMV during a particular handshake. However, the TNCS is not required to do this.

If an IMV does not provide a recommendation earlier, the TNCS will call the IMV's solicitRecommendation method at the end of an Integrity Check Handshake (after all IMC-IMV messages have been delivered). The IMV SHOULD then call the IMVConnection's provideRecommendation method to deliver its recommendation. If the IMV calls this method when there is no active handshake on the specified network connection, the TNCS SHOULD throw the TNC_RESULT_ILLEGAL_OPERATION exception. If an IMV really needs to communicate a recommendation at another time, it should call the IMVConnection's requestHandshakeRetry method.

Parameters:
`recommendation` - action recommendation
`evaluation` - evaluation result
Throws:
`TNCException`

`java.lang.Object` **`getAttribute`**`(long attributeID)`
`throws TNCException`

An IMV calls this method to get the value of the attribute identified by `attributeID` for this `IMVConnection`.

This function is optional. The TNCS is not required to implement it. If it is not implemented for this IMVConnection, it MUST throw an UnsupportedOperationException. IMVs MUST work properly if a TNCS does not implement this function.

The IMV MUST pass a standard or vendor-specific attribute ID as the attributeID parameter. If the TNCS does not recognize the attribute ID, it SHOULD throw a TNCException with the TNC_RESULT_INVALID_PARAMETER result code. If the TNCS recognizes the attribute ID but does not have an attribute value for the requested attribute ID for this IMVConnection, it SHOULD also throw a TNCException with the TNC_RESULT_INVALID_PARAMETER result code.

The return value is an Object that represents the attribute value requested. The IMV must cast this Object to the class documented in the description of that specific attribute to get the desired value. All Objects returned by this method SHOULD be immutable.

Parameters:
`attributeID` - the attribute ID of the desired attribute
Returns:
the attribute value
Throws:
`TNCException`

```
void setAttribute(long attributeID,
                  java.lang.Object attributeValue)
                  throws TNCException
```

An IMV calls this method to set the value of the attribute identified by `attributeID` for this `IMVConnection`.

This function is optional. The TNCS is not required to implement it. If it is not implemented for this IMVConnection, it MUST throw an UnsupportedOperationException. IMVs MUST work properly if a TNCS does not implement this function.

The IMV MUST pass a standard or vendor-specific attribute ID as the attributeID parameter. If the TNCS does not recognize the attribute ID, it SHOULD throw a TNCException with the TNC_RESULT_INVALID_PARAMETER result code. If the TNCS recognizes the attribute ID but does not support setting an attribute value for the requested attribute ID for this IMVConnection, it SHOULD also throw a TNCException with the TNC_RESULT_INVALID_PARAMETER result code.

For the attributeValue parameter, the IMV MUST pass an Object that represents the new attribute value (or null if permitted for the specified attribute). This Object must actually be an instance of the class documented in the description of the specified attribute. The Object SHOULD be immutable. If the TNCS has any uncertainty about it SHOULD copy the object. The TNCS MAY check the Object and throw a TNCException if it is not a valid value for the specified attribute.

Parameters:
`attributeID` - the attribute ID of the attribute to be set
`attributeValue` - the new value to be set for this attribute
Throws:
`TNCException`

4.3.12.10 IMVConnectionLong Interface (IMVConnectionLong.java)

org.trustedcomputinggroup.tnc.ifimv

Interface IMVConnectionLong

`public interface` **`IMVConnectionLong`**
This interface can be implemented by an object that already supports the IMVConnection interface to indicate that this object is an IMVConnection that supports sendMessageLong. To check whether an object implements this interface, use the `instanceof` operator. If so, cast the object to the interface type and use the methods in this interface.

Method Detail

```
void sendMessageLong(long messageFlags,
                     long messageVendorID,
                     long messageSubtype,
                     byte[] message,
                     long sourceIMVID,
                     long destinationIMCID)
                     throws TNCException
```

An IMV calls this method to give a message to the TNCS for delivery. This method provides several features that `sendMessage` does not: longer 32 bit vendor ID and message subtype fields, a `messageFlags` field, and the ability to specify a source IMV ID and a desired or exclusive destination IMC ID.

The message is contained in the buffer referenced by `message`. This parameter may be null to represent an empty message. The type of the message is indicated by the `messageVendorID` and `messageSubtype` parameters. TNCSs MAY check these values to make sure they are valid and throw an exception if not, but TNCSs are not required to make these checks.

The `messageFlags` parameter supports up to 32 flags that may be set pertaining to the message. At this time, only one flag has been defined. The value 0x80000000 is known as the `TNC_MESSAGE_FLAGS_EXCLUSIVE` flag ("the EXCL flag", for short). If this bit is set, the sending IMV is requesting that the message only be delivered to the IMC designated by the `destinationIMCID` parameter. If the bit is cleared, the message is to be delivered to any IMC that has indicated an interest in the message type. Other flags may be defined in future versions of this specification. Until that time, IMVs MUST leave all these flags cleared. Some connections and/or TNCSs may not support the EXCL flag. To determine whether a particular TNCS and connection supports this flag, the IMV should get the Has Exclusive attribute for that connection. If the IMV sets a flag that is not supported by the TNCS or the connection, the TNCS SHOULD throw a `TNCException` with a `TNC_RESULT_INVALID_PARAMETER` result.

The IMV MUST set the `sourceIMVID` parameter to either the primary IMV ID queried by IMV using the `TNC_AttributeID_PRIMARY_IMV_ID` attribute, or an additional IMV ID reserved when IMV requests the TNCS to do so by calling `reserveAdditionalIMVID`. This IMV ID will be used as the sourceIMVID for protocols that support sending this field along with the message.

If the EXCL flag is set, the IMV MUST set the `destinationIMCID` parameter to indicate which IMC should receive the message being sent. The IMV should place into this parameter a value which it has obtained using the `sourceIMCID` parameter for messages previously delivered to the IMV on this connection. The IMV MAY set the `destinationIMCID` parameter to a specific IMC ID even when the EXCL flag is not set. This indicates that the message being sent is in response to a message received from the IMC indicated in the `destinationIMCID` parameter. If the IMV does not know the IMC ID of the recipient IMC, the IMV MUST use the `TNC_IMCID_ANY` wild card for the `destinationIMCID` parameter and MUST NOT set the EXCL flag. In such case, the message is delivered to all IMCs that have expressed interest in receiving such messages. This may happen when the TNCS starts the integrity handshake and the IMV does not know the IMC IDs of the recipient IMCs.

TNCSs are not required to implement this method, but any TNCS that implements the IMVConnectionLong interface MUST implement this method. Since this method was not included in IF-IMV 1.0 through 1.2, many TNCSs do not implement it. IMVs MUST work properly if a TNCS does not implement this method. The IMV is never required to call this method. The TNCS MUST work with IMVs that don't call this method. The IMV also SHOULD check the Has Long attribute for a given connection, to determine whether this connection supports long types, before calling this method for that connection. If the connection does not support long types, the IMV MAY call this method (if implemented) but MUST NOT use a `messageVendorID` greater than `0xffffff` or a `messageSubtype` greater than `0xff`. If the IMV does so, the TNCS SHOULD throw a `TNC_RESULT_NO_LONG_TYPES` exception.

The TNC Server MUST NOT ever modify the buffer contents and MUST NOT access the buffer after `sendMessageLong` has returned. The TNC Server will typically copy the message out of the buffer, queue it up for delivery, and return from this method.

The IMV MUST NOT call this method unless it has received a call to `beginHandshake`, `receiveMessage`, `receiveMessageSOH`, `receiveMessageLong`, or `batchEnding` for this connection and has not yet returned from that method. If the IMV violates this prohibition, the TNCS SHOULD throw a `TNC_RESULT_ILLEGAL_OPERATION` exception. If an IMV really wants to communicate with an IMC at another time, it should call `requestHandshakeRetry`.

Note that a TNCC or TNCS MAY cut off IMC-IMV communications at any time for any reason, including limited support for long conversations in underlying protocols, user or administrator intervention, or policy. If this happens, the TNCS will throw a `TNC_RESULT_ILLEGAL_OPERATION` exception from `sendMessageLong`. If the TNCS supports limiting the message size or number of round trips, the TNCS MUST throw a `TNC_RESULT_EXCEEDED_MAX_MESSAGE_SIZE` or `TNC_RESULT_EXCEEDED_MAX_ROUND_TRIPS` exception respectively if the limits are exceeded. An IMV can get the Maximum Message Size and Maximum Number of Round Trips by using the related Attribute ID within the `getAttribute` method. The IMV SHOULD adapt its behavior to accommodate these limitations if available.

The TNC Server MUST support any message type. However, the IMV MUST NOT specify a message type whose vendor ID is 0xffffff or whose subtype is 0xff. These values are reserved for use as wild cards, as described in section 3.9.1. If the IMV violates this prohibition, the TNCS SHOULD throw a `TNC_RESULT_INVALID_PARAMETER` exception.

Parameters:
`messageFlags` – flags associated with the message
`messageVendorID` – vendor ID associated with the message
`messageSubtype` – message subtype associated with the message
`message` – the message to be delivered
`sourceIMVID` – source IMVID for the message
`destinationIMCID` – destination IMCID for the message
Throws:
`TNCException` - if a TNC error occurs

4.3.12.11 IMVConnectionSOH Interface (IMVConnectionSOH.java)

org.trustedcomputinggroup.tnc.ifimv

Interface IMVConnectionSOH

`public interface` **`IMVConnectionSOH`**
This interface can be implemented by an object that already supports the IMVConnection interface to indicate that this object is an IMVConnection that supports sendMessageSOH. To check whether an object implements this interface, use the `instanceof` operator. If so, cast the object to the interface type and use the methods in this interface.

Method Detail

`void` **`sendMessageSOH`**`(byte[] sohrReportEntry)`
`throws TNCException`

An IMV calls this method (if implemented by the TNCS) to give a SoHRReportEntry to the TNCS for delivery. This allows an IMV to send an entire SoHRReportEntry instead of just the contents of the Vendor-Specific attribute, as would be the case if the `sendMessage` method was used.

The SoHRReportEntry (as defined in section 3.5 of [12]) is contained in the buffer referenced by the `sohrReportEntry` parameter. If the length of the `sohrReportEntry` parameter is zero (0), the TNCS MUST throw a `TNC_RESULT_INVALID_PARAMETER` exception, since a zero-length SoHRReportEntry is prohibited by the IF_TNCCS-SOH 1.0 specification. No type is included with the SoHRReportEntry, since this is included in the System-Health-ID attribute in the SoHRReportEntry. TNCSs MAY check these values to make sure they are valid and throw a `TNC_RESULT_INVALID_PARAMETER` exception if not, but TNCSs are not required to make these checks.

A TNCS that supports IF-TNCCS-SOH SHOULD support this method. The TNC Server MUST NOT ever modify the buffer contents and MUST NOT access the buffer after `sendMessageSOH` has returned. The TNC Server will typically copy the SoHRReportEntry out of the buffer, place it into the SoHR message or queue it up for such placement, and return from this method.

An IMV may call this method if it needs to send an SoHRReportEntry with a connection that supports IF-TNCCS-SOH (as indicated by the Has SOH attribute being set to a value of 1). If an IMV calls this method for a connection that does not support IF-TNCCS-SOH, the TNCS SHOULD throw a `TNC_RESULT_NO_SOH_SUPPORT` exception from `sendMessageSOH`. IMVs should recognize that many TNCSs do not support IF-TNCCS-SOH, and some TNCSs that do support IF-TNCCS-SOH will not support this method. Therefore, IMVs should be prepared to send messages via `sendMessage` in some cases when IF-TNCC-SOH is used. Simple IMVs that only need to send simple messages need not call this method. They can just call `sendMessage`, which will automatically place their message in a Vendor-Specific attribute in an SoHRReportEntry.

The IMV MUST NOT call this method unless it has received a call to `beginHandshake`, `receiveMessage`, `receiveMessageSOH`, `receiveMessageLong`, or `batchEnding` for this connection and has not yet returned from that method. If the IMV violates this prohibition, the TNCS SHOULD throw a `TNC_RESULT_ILLEGAL_OPERATION` exception. If an IMV really wants to communicate with an IMC at another time, it should call `requestHandshakeRetry`.

Note that a TNCC or TNCS MAY cut off IMC-IMV communications at any time for any reason, including limited support for long conversations in underlying protocols, user or administrator intervention, or policy. If this happens, the TNCS will throw a

`TNC_RESULT_ILLEGAL_OPERATION` exception from `sendMessageSOH` and may call `solicitRecommendation` to elicit IMV Action Recommendations based on the data they have gathered so far.

If the TNCS supports limiting the message size or number of round trips, and the limits are exceeded, the TNCS MUST throw a `TNC_RESULT_EXCEEDED_MAX_MESSAGE_SIZE` or `TNC_RESULT_EXCEEDED_MAX_ROUND_TRIPS` exception, as the case may be. An IMV can get the Maximum Message Size and Maximum Number of Round Trips by using the related Attribute ID within the getAttribute method. The IMV SHOULD adapt its behavior to accommodate these limitations if available.

Parameters:
`sohrReportEntry` - sohrReportEntry to be delivered
Throws:
`TNCException` - if a TNC error occurs

5 Security Considerations

This section describes the security threats related to IF-IMV and suggests methods to address these threats. The components involved in IF-IMV are one or more Trusted Network Communications Servers (TNCS) and one or more Integrity Measurement Verifiers (IMVs). These are logical components; the TNCS and IMVs reside on the same host. The IF-IMV is the interface between the TNCS and the IMVs.

A multitude of remote distributed endpoints is often more difficult to manage securely than a small number of centralized servers; therefore, it is highly recommended that IMV and TNCS implementers read and understand the Security Considerations of the IF-IMC [9] in addition to the considerations in this document.

5.1 Threat Analysis

5.1.1 Registration and Discovery Based Threats

The TNCS discovers which IMVs can be loaded on a host via a platform-specific binding, for example, on the Windows platform using a windows registry key and on the Linux or Unix platform using a configuration file. On Windows, this implies the registry keys are typically created when the IMVs are installed, requiring the IMV installer to possess sufficient privileges on the platform. Similarly the TNCS must have sufficient privileges to read the relevant keys. Based on the IMVs discovered in the registry, the TNCS loads the code referenced by the registry entries. On Linux and UNIX, analogous privilege requirements apply for accessing the configuration file. Any party with sufficient privileges to modify the relevant registry key or configuration file can mount the following attacks on the registration process:

- It can add an invalid IMV (Spoofing)
- It can remove a valid IMV, perhaps replacing it with rogue/modified versions of code (Tamper)

Similar attacks can also be mounted by modifying the code of an IMV or critical data upon which the IMV depends.

The ability to add an invalid IMV can have considerable impact, as detailed in the next section.

5.1.2 Rogue IMV Threats

If a rogue IMV is installed and then loaded by a valid TNCS, it may be able to misuse IF-IMV in the following ways:

- Overwrite TNCS or IMV memory
- Violate IF-IMV API requirements such as passing illegal or unexpected argument values
- Perform illegal operations so that the TNCS is terminated by the operating system
- Perform improper operations with the TNCS' privileges
- Attack other components (such as the NAA or remote servers) using the privileges or credentials of the TNCS or other IMVs
- Send invalid messages to IMCs or IMVs, leading to IMC or IMV crashes or compromise, excessive IMC or IMV resource consumption, or unauthorized or malicious remediation
- Monitor IMC-IMV messages and disclose them or use them for attacks on the AR ("IMV Spyware").
- Issue a large number of, or particularly expensive, interface API calls to the TNCS (Denial of service of the TNCS)
- Provide incorrect IMV Action Recommendations, causing valid clients to be rejected or invalid clients to be let on the network
- Provide incorrect IMV Evaluation Results, causing the system state to not reflect the true compliance state of the endpoint
- Spoof TNCS calls to an IMV and provide incorrect handshake or compliance data to IMVs
- Spuriously request handshake retries (Denial of service)

- Lock up TNCS threads by not returning from function calls (Denial of service)
- Use vendor-specific extensions to IF-IMV to perform other attacks

5.1.3 Rogue TNCS Threats

If a rogue TNCS loads a valid IMV, it may be able to misuse IF-IMV in the following ways:

- Overwrite IMV memory
- Violate IF-IMV API requirements such as passing illegal or unexpected argument values
- Attack other components (such as the NAA or remote servers) using the credentials of an IMV
- Send invalid messages to IMCs or IMVs, leading to IMC or IMV crashes or compromise, excessive IMC or IMV resource consumption, or unauthorized or malicious remediation
- Monitor IMC-IMV messages and disclose them or use them for attacks on the AR
- Issue a large number of, or particularly expensive, interface API calls to an IMV, possibly causing denial of service of a remote server
- Provide incorrect TNCS Action Recommendations to a NAA, causing valid clients to be rejected or invalid clients to be let on the network
- Spuriously request or perform handshake retries (Denial of service)
- Use vendor-specific extensions to IF-IMV to perform other attacks

5.1.4 Man-in-the-Middle Threats

If an attacker injects a man-in-the-middle between an IMV and its corresponding IMC peer entity, between an IMV and its corresponding TNCS, or between a TNCS and a TNCC it may be able to be misused in the following ways:

- Allows the viewing, modification, deletion, or addition, of messages passing between the IMV and the IMC,between the IMV and its corresponding TNCS, or between the TNCS and the TNCC,
- Allow the replay of measurements or other messages that are not reflective of the Access Requestor's current conditions.

5.1.5 Tampering Threats on IMVs and TNCSs

Malicious code (worms, viruses, etc) or another unauthorized application can modify an IMV or TNCS. This allows the attacker to misuse TNC components in the following ways:

- Modify legitimate messages, add new illegitimate messages, or delete legitimate messages.
- Allow the attack to exfiltrate measurements and other data from an Access Requestor.

5.1.6 Threats Beyond IF-IMV

IF-IMV is part of the larger TNC architecture. Successful attacks against other parts of the TNC architecture will generally result in negative effects for IMVs, TNCSs, and the system as a whole. See the Security Considerations section of the TNC Architecture document for an analysis of considerations that pertain to other parts of the TNC architecture.

5.2 Suggested remedies

As demonstrated by the attacks listed above, it is critical that only authorized IMVs be loaded by a TNCS and only authorized TNCSs be allowed to load an IMV. There are well known methods to control what code is loaded by a TNCS:

- Generate a cryptographic hash on the code image and verify it against a list of good hashes
- Verify the software publisher using certificates
- Control access to the IMV registration mechanism (registry or configuration file)
- Control access to IMV code and critical data files
- Employ a TNCS-specific list of authorized IMVs

Similar checks can be performed by the operating system before loading the TNCS.

Industry standard best practices for secure coding, software engineering, and code reviews should be followed during the development of IMVs and TNCSs in order to minimize the possibility of incorrect design, incorrect implementation, and software flaws thus mitigating some of the attacks described above.

The addition of a Platform Trust Service (PTS) may provide the above listed services and may also use hardware such as the Trusted Platform Module (TPM) to establish a trusted load path on a platform which is rooted in hardware. In short, every loader entity on the platform is measured before it loads another component, and the measured loaders are expected to log their measurements with corresponding verification signatures in the TPM. In addition, using PTS for dynamic measuring of TNC components during runtime and also mitigate the attacks related to tampering.

Information disclosure attacks can be prevented by creating security associations between IMCs and IMVs. This does not preclude an additional security association between a NAR and a NAA.

To prevent/detect denial of service attacks, API usage from registered IMVs can be monitored.

"IMV spyware" attacks can often be prevented administratively; for example, by prohibiting unknown programs from making unauthorized network connections, or by monitoring the disk for log files created by unknown IMVs which are simply logging messages.
Note that invalid handshake retries can be mitigated by only allowing a retry on a valid session that is associated with each particular IMV ID.

This specification requires that all valid IMVs be installed to a protected system directory. The loading of a rogue IMV can be mitigated (not prevented) by requiring privileged access to the registry key or config file. Note, however, that some (usually legacy) operating systems have no concept of a "protected" directory, registry, or file, and thus are provided no protection from this scenario. Note that this approach requires best practices for the use of protected directories and registries; if a user has any administrative access to these objects, they are vulnerable to a social engineering approach to causing a Trojan IMV to be installed.

Further protection against rogue IMVs (and also against buggy IMVs) can be provided by having the TNCS launch a new "child" process for each IMV, having the child process load the IMV, and then having the TNCS communicate with the child processes carefully. This limits the amount of damage that can be done by a rogue IMV. The TNCS may use this approach but is not required to do so.

IMV implementers who choose a stub-to-backend-server implementation must take care not to make the stub-to-server communications the "weak link" in the security chain. They should choose protocols which maintain integrity and confidentiality as required, while taking into account the need for efficiency.

One countermeasure for a man-in-the-middle attack is to make use of the PTS described in this section earlier. An additional countermeasure is to have the IF-M protocol (between the IMV and the IMC) and/or the IF-TNCCS (between the TNCS and the TNCC) provide both strong mutual authentication and anti-replay technology in a similar manner to the IF-T protocol. Note: Future enhancements to this specification may be necessary for this type of counter measure.

Protection from many of the identified threats can be provided by housing the IMVs and TNCSs separately from that which is being measured. If a particular component exists within an isolated environment, the chance of it being compromised is far reduced. It is recommended that careful

analysis of the threat environment that a TNC implementation will be deployed into be conducted and the strongest such isolation that makes sense in that environment be applied.

6 C Header File

This section provides a C header file that serves as a binding for the IF-IMV API with the C language and the Microsoft Windows DLL platform binding. As noted in section 3.1, implementers SHOULD use the C language binding when possible for maximum compatibility with other IMVs and TNC Servers on their platform.

```
/* tncifimv.h
 *
 * Trusted Network Communications IF-IMV API version 1.40
 * Microsoft Windows DLL Platform Binding C Header
 * June 3, 2013
 *
 * Copyright(c) 2005-2013, Trusted Computing Group, Inc. All rights
 * reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * - Redistributions of source code must retain the above copyright
 *   notice, this list of conditions and the following disclaimer.
 * - Redistributions in binary form must reproduce the above copyright
 *   notice, this list of conditions and the following disclaimer in
 *   the documentation and/or other materials provided with the
 *   distribution.
 * - Neither the name of the Trusted Computing Group nor the names of
 *   its contributors may be used to endorse or promote products
 *   derived from this software without specific prior written
 *   permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
 * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
 * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
 * FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
 * COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
 * INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING,
 * BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
 * LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
 * CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN
 * ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
 * POSSIBILITY OF SUCH DAMAGE.
 *
 * Contact the Trusted Computing Group at
 * admin@trustedcomputinggroup.org for information on specification
 * licensing through membership agreements.
 *
 * Any marks and brands contained herein are the property of their
 * respective owners.
 */

#ifndef _TNCIFIMV_H
#define _TNCIFIMV_H

#ifdef __cplusplus
extern "C" {
#endif
```

```
#ifdef WIN32
#ifdef TNC_IMV_EXPORTS
#define TNC_IMV_API __declspec(dllexport)
#else
#define TNC_IMV_API __declspec(dllimport)
#endif
#else
#define TNC_IMV_API
#endif

/* Basic Types */

typedef unsigned long TNC_UInt32;
typedef unsigned char *TNC_BufferReference;

/* Derived Types */

typedef TNC_UInt32 TNC_IMVID;
typedef TNC_UInt32 TNC_ConnectionID;
typedef TNC_UInt32 TNC_ConnectionState;
typedef TNC_UInt32 TNC_RetryReason;
typedef TNC_UInt32 TNC_IMV_Action_Recommendation;
typedef TNC_UInt32 TNC_IMV_Evaluation_Result;
typedef TNC_UInt32 TNC_MessageType;
typedef TNC_MessageType *TNC_MessageTypeList;
typedef TNC_UInt32 TNC_VendorID;
typedef TNC_VendorID *TNC_VendorIDList;
typedef TNC_UInt32 TNC_MessageSubtype;
typedef TNC_MessageSubtype *TNC_MessageSubtypeList;
typedef TNC_UInt32 TNC_Version;
typedef TNC_UInt32 TNC_Result;
typedef TNC_UInt32 TNC_AttributeID;

/* Function pointers */

typedef TNC_Result (*TNC_IMV_InitializePointer)(
    TNC_IMVID imvID,
    TNC_Version minVersion,
    TNC_Version maxVersion,
    TNC_Version *pOutActualVersion);
typedef TNC_Result (*TNC_IMV_NotifyConnectionChangePointer)(
    TNC_IMVID imvID,
    TNC_ConnectionID connectionID,
    TNC_ConnectionState newState);
typedef TNC_Result (*TNC_IMV_ReceiveMessagePointer)(
    TNC_IMVID imvID,
    TNC_ConnectionID connectionID,
    TNC_BufferReference message,
    TNC_UInt32 messageLength,
    TNC_MessageType messageType);
typedef TNC_Result (*TNC_IMV_ReceiveMessageSOHPointer)(
    TNC_IMVID imvID,
    TNC_ConnectionID connectionID,
    TNC_BufferReference sohReportEntry,
    TNC_UInt32 sohRELength,
    TNC_MessageType systemHealthID);
```

```
typedef TNC_Result (*TNC_IMV_ReceiveMessageLongPointer)(
    TNC_IMVID imvID,
    TNC_ConnectionID connectionID,
    TNC_UInt32 messageFlags,
    TNC_BufferReference message,
    TNC_UInt32 messageLength,
    TNC_VendorID messageVendorID,
    TNC_MessageSubtype messageSubtype,
    TNC_UInt32 sourceIMCID,
    TNC_UInt32 destinationIMVID);
typedef TNC_Result (*TNC_IMV_SolicitRecommendationPointer)(
    TNC_IMVID imvID,
    TNC_ConnectionID connectionID);
typedef TNC_Result (*TNC_IMV_BatchEndingPointer)(
    TNC_IMVID imvID,
    TNC_ConnectionID connectionID);
typedef TNC_Result (*TNC_IMV_TerminatePointer)(
    TNC_IMVID imvID);
typedef TNC_Result (*TNC_TNCS_ReportMessageTypesPointer)(
    TNC_IMVID imvID,
    TNC_MessageTypeList supportedTypes,
    TNC_UInt32 typeCount);
typedef TNC_Result (*TNC_TNCS_ReportMessageTypesLongPointer)(
    TNC_IMVID imvID,
    TNC_VendorIDList supportedVendorIDs,
    TNC_MessageSubtypeList supportedSubtypes,
    TNC_UInt32 typeCount);
typedef TNC_Result (*TNC_TNCS_SendMessagePointer)(
    TNC_IMVID imvID,
    TNC_ConnectionID connectionID,
    TNC_BufferReference message,
    TNC_UInt32 messageLength,
    TNC_MessageType messageType);
typedef TNC_Result (*TNC_TNCS_SendMessageSOHPointer)(
    TNC_IMVID imvID,
    TNC_ConnectionID connectionID,
    TNC_BufferReference sohrReportEntry,
    TNC_UInt32 sohrRELength);
typedef TNC_Result (*TNC_TNCS_SendMessageLongPointer)(
    TNC_IMVID imvID,
    TNC_ConnectionID connectionID,
    TNC_UInt32 messageFlags,
    TNC_BufferReference message,
    TNC_UInt32 messageLength,
    TNC_VendorID messageVendorID,
    TNC_MessageSubtype messageSubtype,
    TNC_UInt32 destinationIMCID);
typedef TNC_Result (*TNC_TNCS_RequestHandshakeRetryPointer)(
    TNC_IMVID imvID,
    TNC_ConnectionID connectionID,
    TNC_RetryReason reason);
typedef TNC_Result (*TNC_TNCS_ProvideRecommendationPointer)(
    TNC_IMVID imvID,
    TNC_ConnectionID connectionID,
    TNC_IMV_Action_Recommendation recommendation,
    TNC_IMV_Evaluation_Result evaluation);
typedef TNC_Result (*TNC_TNCS_GetAttributePointer)(
```

```
    TNC_IMVID imvID,
    TNC_ConnectionID connectionID,
    TNC_AttributeID attributeID,
    TNC_UInt32 bufferLength,
    TNC_BufferReference buffer,
    TNC_UInt32 *pOutValueLength);
typedef TNC_Result (*TNC_TNCS_SetAttributePointer)(
    TNC_IMVID imvID,
    TNC_ConnectionID connectionID,
    TNC_AttributeID attributeID,
    TNC_UInt32 bufferLength,
    TNC_BufferReference buffer);
typedef TNC_Result (*TNC_TNCS_ReserveAdditionalIMVIDPointer)(
    TNC_IMVID imvID,
    TNC_UInt32 *pOutIMVID);
typedef TNC_Result (*TNC_TNCS_BindFunctionPointer)(
    TNC_IMVID imvID,
    char *functionName,
    void **pOutfunctionPointer);
typedef TNC_Result (*TNC_IMV_ProvideBindFunctionPointer)(
    TNC_IMVID imvID,
    TNC_TNCS_BindFunctionPointer bindFunction);

/* Result Codes */

#define TNC_RESULT_SUCCESS 0
#define TNC_RESULT_NOT_INITIALIZED 1
#define TNC_RESULT_ALREADY_INITIALIZED 2
#define TNC_RESULT_NO_COMMON_VERSION 3
#define TNC_RESULT_CANT_RETRY 4
#define TNC_RESULT_WONT_RETRY 5
#define TNC_RESULT_INVALID_PARAMETER 6
/* reserved for TNC_RESULT_CANT_RESPOND: 7 */
#define TNC_RESULT_ILLEGAL_OPERATION 8
#define TNC_RESULT_OTHER 9
#define TNC_RESULT_FATAL 10
#define TNC_RESULT_EXCEEDED_MAX_ROUND_TRIPS 0x00559700
#define TNC_RESULT_EXCEEDED_MAX_MESSAGE_SIZE 0x00559701
#define TNC_RESULT_NO_LONG_MESSAGE_TYPES 0x00559702
#define TNC_RESULT_NO_SOH_SUPPORT 0x00559703



/* Version Numbers */

#define TNC_IFIMV_VERSION_1 1

/* Network Connection ID Values */

#define TNC_CONNECTIONID_ANY 0xFFFFFFFF

/* Network Connection State Values */

#define TNC_CONNECTION_STATE_CREATE 0
#define TNC_CONNECTION_STATE_HANDSHAKE 1
#define TNC_CONNECTION_STATE_ACCESS_ALLOWED 2
#define TNC_CONNECTION_STATE_ACCESS_ISOLATED 3
```

```
#define TNC_CONNECTION_STATE_ACCESS_NONE 4
#define TNC_CONNECTION_STATE_DELETE 5

/* Handshake Retry Reason Values */

/* reserved for TNC_RETRY_REASON_IMC_REMEDIATION_COMPLETE: 0 */
/* reserved for TNC_RETRY_REASON_IMC_SERIOUS_EVENT: 1 */
/* reserved for TNC_RETRY_REASON_IMC_INFORMATIONAL_EVENT: 2 */
/* reserved for TNC_RETRY_REASON_IMC_PERIODIC: 3 */
#define TNC_RETRY_REASON_IMV_IMPORTANT_POLICY_CHANGE 4
#define TNC_RETRY_REASON_IMV_MINOR_POLICY_CHANGE 5
#define TNC_RETRY_REASON_IMV_SERIOUS_EVENT 6
#define TNC_RETRY_REASON_IMV_MINOR_EVENT 7
#define TNC_RETRY_REASON_IMV_PERIODIC 8

/* IMV Action Recommendation Values */

#define TNC_IMV_ACTION_RECOMMENDATION_ALLOW 0
#define TNC_IMV_ACTION_RECOMMENDATION_NO_ACCESS 1
#define TNC_IMV_ACTION_RECOMMENDATION_ISOLATE 2
#define TNC_IMV_ACTION_RECOMMENDATION_NO_RECOMMENDATION 3

/* IMV Evaluation Result Values */

#define TNC_IMV_EVALUATION_RESULT_COMPLIANT 0
#define TNC_IMV_EVALUATION_RESULT_NONCOMPLIANT_MINOR 1
#define TNC_IMV_EVALUATION_RESULT_NONCOMPLIANT_MAJOR 2
#define TNC_IMV_EVALUATION_RESULT_ERROR 3
#define TNC_IMV_EVALUATION_RESULT_DONT_KNOW 4

/* IMC/IMV ID Values */

#define TNC_IMVID_ANY ((TNC_UInt32) 0xffff)
#define TNC_IMCID_ANY ((TNC_UInt32) 0xffff)

/* Vendor ID Values */

#define TNC_VENDORID_TCG 0
#define TNC_VENDORID_TCG_NEW 0x005597
#define TNC_VENDORID_ANY ((TNC_VendorID) 0xffffff)

/* Message Subtype Values */

#define TNC_SUBTYPE_ANY ((TNC_MessageSubtype) 0xff)

/* Message Flags Values */

#define TNC_MESSAGE_FLAGS_EXCLUSIVE ((TNC_UInt32) 0x80000000)

/* Message Attribute ID Values */

#define      TNC_ATTRIBUTEID_PREFERRED_LANGUAGE      ((TNC_AttributeID)
0x00000001)
#define TNC_ATTRIBUTEID_REASON_STRING ((TNC_AttributeID) 0x00000002)
#define TNC_ATTRIBUTEID_REASON_LANGUAGE ((TNC_AttributeID) 0x00000003)
#define TNC_ATTRIBUTEID_MAX_ROUND_TRIPS ((TNC_AttributeID) 0x00559700)
#define TNC_ATTRIBUTEID_MAX_MESSAGE_SIZE ((TNC_AttributeID) 0x00559701)
```

```
#define TNC_ATTRIBUTEID_DHPN ((TNC_AttributeID) 0x00559702)
#define TNC_ATTRIBUTEID_HAS_LONG_TYPES  ((TNC_AttributeID) 0x00559703)
#define TNC_ATTRIBUTEID_HAS_EXCLUSIVE ((TNC_AttributeID) 0x00559704)
#define TNC_ATTRIBUTEID_HAS_SOH ((TNC_AttributeID) 0x00559705)
#define TNC_ATTRIBUTEID_SOH ((TNC_AttributeID) 0x00559706)
#define TNC_ATTRIBUTEID_SSOH ((TNC_AttributeID) 0x00559707)
#define TNC_ATTRIBUTEID_IFTNCCS_PROTOCOL ((TNC_AttributeID) 0x0055970A)
#define TNC_ATTRIBUTEID_IFTNCCS_VERSION ((TNC_AttributeID) 0x0055970B)
#define TNC_ATTRIBUTEID_IFT_PROTOCOL ((TNC_AttributeID) 0x0055970C)
#define TNC_ATTRIBUTEID_IFT_VERSION ((TNC_AttributeID) 0x0055970D)
#define TNC_ATTRIBUTEID_TLS_UNIQUE ((TNC_AttributeID) 0x0055970E)
#define TNC_ATTRIBUTEID_PRIMARY_IMV_ID ((TNC_AttributeID) 0x00559710)
#define TNC_ATTRIBUTEID_AR_IDENTITIES ((TNC_AttributeID) 0x00559712)

/* TCG Standard Identity Types */

#define TNC_ID_UNKNOWN ((TNC_UInt32) 0x00000000)
#define TNC_ID_IPV4_ADDR ((TNC_UInt32) 0x00000001)
#define TNC_ID_IPV6_ADDR ((TNC_UInt32) 0x00000002)
#define TNC_ID_FQDN ((TNC_UInt32) 0x00000003)
#define TNC_ID_EMAIL_ADDR ((TNC_UInt32) 0x00000004)
#define TNC_ID_USERNAME ((TNC_UInt32) 0x00000005)
#define TNC_ID_X500_DN ((TNC_UInt32) 0x00000006)

/* TCG Standard Subject Types */

#define TNC_SUBJECT_UNKNOWN ((TNC_UInt32) 0x00000000)
#define TNC_SUBJECT_MACHINE ((TNC_UInt32) 0x00000001)
#define TNC_SUBJECT_USER ((TNC_UInt32) 0x00000002)

/* TCG Standard Authentication Methods */

#define TNC_AUTH_UNKNOWN ((TNC_UInt32) 0x00000000)
#define TNC_AUTH_X509_CERT ((TNC_UInt32) 0x00000001)
#define TNC_AUTH_PASSWORD ((TNC_UInt32) 0x00000002)
#define TNC_AUTH_SIM ((TNC_UInt32) 0x00000003)

/* IMV Functions */

TNC_IMV_API TNC_Result TNC_IMV_Initialize(
/*in*/  TNC_IMVID imvID,
/*in*/  TNC_Version minVersion,
/*in*/  TNC_Version maxVersion,
/*in*/  TNC_Version *pOutActualVersion);

TNC_IMV_API TNC_Result TNC_IMV_NotifyConnectionChange(
/*in*/  TNC_IMVID imvID,
/*in*/  TNC_ConnectionID connectionID,
/*in*/  TNC_ConnectionState newState);

TNC_IMV_API TNC_Result TNC_IMV_ReceiveMessage(
/*in*/  TNC_IMVID imvID,
/*in*/  TNC_ConnectionID connectionID,
/*in*/  TNC_BufferReference messageBuffer,
/*in*/  TNC_UInt32 messageLength,
/*in*/  TNC_MessageType messageType);
```

```
TNC_IMV_API TNC_Result TNC_IMV_ReceiveMessageSOH(
/*in*/  TNC_IMVID imvID,
/*in*/  TNC_ConnectionID connectionID,
/*in*/  TNC_BufferReference sohReportEntry,
/*in*/  TNC_UInt32 sohRELength,
/*in*/  TNC_MessageType systemHealthID);

TNC_IMV_API TNC_Result TNC_IMV_ReceiveMessageLong(
/*in*/  TNC_IMVID imvID,
/*in*/  TNC_ConnectionID connectionID,
/*in*/  TNC_UInt32 messageFlags,
/*in*/  TNC_BufferReference message,
/*in*/  TNC_UInt32 messageLength,
/*in*/  TNC_VendorID messageVendorID,
/*in*/  TNC_MessageSubtype messageSubtype,
/*in*/  TNC_UInt32 sourceIMCID,
/*in*/  TNC_UInt32 destinationIMVID);

TNC_IMV_API TNC_Result TNC_IMV_SolicitRecommendation(
/*in*/  TNC_IMVID imvID,
/*in*/  TNC_ConnectionID connectionID);

TNC_IMV_API TNC_Result TNC_IMV_BatchEnding(
/*in*/  TNC_IMVID imvID,
/*in*/  TNC_ConnectionID connectionID);

TNC_IMV_API TNC_Result TNC_IMV_Terminate(
/*in*/  TNC_IMVID imvID);

TNC_IMV_API TNC_Result TNC_IMV_ProvideBindFunction(
/*in*/  TNC_IMVID imvID,
/*in*/  TNC_TNCS_BindFunctionPointer bindFunction);

/* TNC Server Functions */

TNC_Result TNC_TNCS_ReportMessageTypes(
/*in*/  TNC_IMVID imvID,
/*in*/  TNC_MessageTypeList supportedTypes,
/*in*/  TNC_UInt32 typeCount);

TNC_Result TNC_TNCS_ReportMessageTypesLong(
/*in*/  TNC_IMVID imvID,
/*in*/  TNC_VendorIDList supportedVendorIDs,
/*in*/  TNC_MessageSubtypeList supportedSubtypes,
/*in*/  TNC_UInt32 typeCount);

TNC_Result TNC_TNCS_SendMessage(
/*in*/  TNC_IMVID imvID,
/*in*/  TNC_ConnectionID connectionID,
/*in*/  TNC_BufferReference message,
/*in*/  TNC_UInt32 messageLength,
/*in*/  TNC_MessageType messageType);

TNC_Result TNC_TNCS_SendMessageSOH(
/*in*/  TNC_IMVID imvID,
/*in*/  TNC_ConnectionID connectionID,
/*in*/  TNC_BufferReference sohrReportEntry,
```

```
/*in*/  TNC_UInt32 sohrRELength);

TNC_Result TNC_TNCS_SendMessageLong(
/*in*/  TNC_IMVID imvID,
/*in*/  TNC_ConnectionID connectionID,
/*in*/  TNC_UInt32 messageFlags,
/*in*/  TNC_BufferReference message,
/*in*/  TNC_UInt32 messageLength,
/*in*/  TNC_VendorID messageVendorID,
/*in*/  TNC_MessageSubtype messageSubtype,
/*in*/  TNC_UInt32 destinationIMCID);

TNC_Result TNC_TNCS_RequestHandshakeRetry(
/*in*/  TNC_IMVID imvID,
/*in*/  TNC_ConnectionID connectionID,
/*in*/  TNC_RetryReason reason);

TNC_Result TNC_TNCS_ProvideRecommendation(
/*in*/  TNC_IMVID imvID,
/*in*/  TNC_ConnectionID connectionID,
/*in*/  TNC_IMV_Action_Recommendation recommendation,
/*in*/  TNC_IMV_Evaluation_Result evaluation);

TNC_Result TNC_TNCS_GetAttribute(
/*in*/  TNC_IMVID imvID,
/*in*/  TNC_ConnectionID connectionID,
/*in*/  TNC_AttributeID attributeID,
/*in*/  TNC_UInt32 bufferLength,
/*out*/ TNC_BufferReference buffer,
/*out*/ TNC_UInt32 *pOutValueLength);

TNC_Result TNC_TNCS_ReserveAdditionalIMVID(
/*in*/  TNC_IMVID imvID,
/*out*/ TNC_UInt32 *pOutIMVID);

TNC_Result TNC_TNCS_SetAttribute(
/*in*/  TNC_IMVID imvID,
/*in*/ TNC_ConnectionID connectionID,
/*in*/ TNC_AttributeID attributeID,
/*in*/ TNC_UInt32 bufferLength,
/*in*/ TNC_BufferReference buffer);
TNC_Result TNC_TNCS_BindFunction(
/*in*/  TNC_IMVID imvID,
/*in*/  char *functionName,
/*in*/  void **pOutfunctionPointer);

#ifdef __cplusplus
}
#endif

#endif
```

7 Use Case Walkthrough

This section provides an informative (non-binding) walkthrough of a typical TNC use case, showing how IF-IMV supports the use case. The text describing IF-IMV usage is in **bold**. Sequence diagrams that illustrate the main parts of this walkthrough are included at the end of this section.

7.1 Configuration

The IT administrator configures any addressing and security information needed for server-side components (PEP, NAA, TNCS, and IMVs) to securely contact each other. The manner in which the PEP, NAA, and TNCS find each other is not specified. The client-side components (TNCC and IMCs) find each other automatically using Microsoft Windows registry or a configuration file modified at install time.

The IT administrator configures policies in the NAA, TNCS, and IMVs for what sorts of user authentication, platform authentication, and integrity checks are required when.

7.2 TNCS Startup

1. When the TNCS starts up, the TNCS loads the IMVs. **[IF-IMV] The details of the load process are platform-specific. With the Microsoft Windows DLL binding, the TNCS reads a protected registry key to find the IMV DLLs, then loads them. During the load process, the TNCS may check the integrity of the IMVs. This is optional.**
2. The TNCS initializes the IMVs through IF-IMV. **[IF-IMV] The TNCS calls `TNC_IMV_Initialize` for each IMV. The IMV performs any initialization it may need to, such as connecting to a remote server process or starting threads. Most IMVs will call `TNC_TNCS_ReportMessageTypes` to indicate which message types they would like to receive. With some platform bindings, this callback must wait until the next step when the Dynamic Function Binding mechanism is functional.**
3. **[IF-IMV] The TNCS performs any other platform-specific initialization needed. With the Microsoft Windows DLL binding, the TNC Server calls the `TNC_IMV_ProvideBindFunction` function to give each IMV a pointer to the bind function (`TNC_TNCS_BindFunction`) used for Dynamic Function Binding.**

7.3 TNCC Startup

1. When the TNCC starts up, the TNCC loads the IMCs.
2. The TNCC initializes the IMCs through IF-IMC.

7.4 Network Connect

1. The endpoint's NAR attempts to connect to a network protected by a PEP, thus triggering an Integrity Check Handshake. There are other ways that an Integrity Check Handshake can be triggered, but this will probably be the most common. For those other ways, the next few steps may be significantly different.
2. The PEP sends a network access decision request to the PDP (NAA or TNCS). Depending on configuration, the PEP may contact the NAA first or the TNCS. The ordering of user authentication, platform authentication, and integrity check is also subject to configuration. Here we present what will probably be the most common order: first user authentication, then platform authentication, then integrity check.
3. The NAA performs user authentication with the NAR. Based on the NAA's policy, the user identity established through this process may be used to make immediate access decisions (like deny). If an immediate access decision has been made, skip to step 17. User authentication may also involve having the NAR authenticate the NAA.

4. The NAA informs the TNCS of the connection request, providing the user identity and other useful info (service requested, etc.).
5. The TNCS performs platform authentication with the TNCC, if required by TNCS policy. This includes verifying the IMC hashes collected during TNCC Setup. If an immediate access decision has been made, skip to step 17. Platform authentication may be mutual so the TNCC can be sure it's talking to a secure server.
6. The TNCC uses IF-IMC to fetch IMC messages.
7. The TNCS uses IF-IMV to inform each IMV that an Integrity Check Handshake has started. **[IF-IMV] If this is a new network connection, the TNCS calls `TNC_IMV_NotifyConnectionChange` with the `newState` parameter set to `TNC_CONNECTION_STATE_CREATE` to indicate that a new network connection has been created. Then the TNCS calls `TNC_IMV_NotifyConnectionChange` with the `newState` parameter set to `TNC_CONNECTION_STATE_HANDSHAKE`.**
8. The TNCC passes the IMC messages to the TNCS. This and all other TNCC-TNCS communications can be sent directly but they will often be relayed through one or more of the NAR, PEP, and NAA.
9. The TNCS passes each IMC message to the matching IMV or IMVs through IF-IMV (using message types associated with the IMC messages to find the right IMV). If there are no IMC messages, skip to step 13. **[IF-IMV] The TNCS delivers the IMC messages to the IMVs by calling `TNC_IMV_ReceiveMessage`. The IMVs may call `TNC_TNCS_SendMessage` before returning from `TNC_IMV_ReceiveMessage` if they want to send a response. When the TNCS has delivered all the IMC messages to the IMVs, it calls `TNC_IMV_BatchEnding` to inform them of this fact. The IMVs may call `TNC_TNCS_SendMessage` before returning from `TNC_IMV_BatchEnding` if they want to send a message to an IMV.**
10. Each IMV analyzes the IMC messages. If an IMV needs to exchange more messages (including remediation instructions) with an IMC, it provides a message to the TNCS and continues with step 11. If an IMV is ready to decide on an IMV Action Recommendation and IMV Evaluation Result, it gives this result to the TNCS through IF-IMV. If there are no more messages to be sent to the IMC from any of the IMVs, skip to step 13. **[IF-IMV] As described in the previous step, IMVs send messages by calling `TNC_TNCS_SendMessage` before returning from `TNC_IMV_ReceiveMessage` and `TNC_IMV_BatchEnding`. IMVs give their results to the TNCS by calling `TNC_TNCS_ProvideRecommendation` at any time.**
11. The TNCS sends the messages from the IMVs to the TNCC.
12. The TNCC sends the IMV messages on to the IMCs through IF-IMC so they can process the messages and respond. Skip to step 8.
13. If there are any IMVs that have not given an IMV Action Recommendation to the TNCS, they are prompted to do so through IF-IMV. **[IF-IMV] The TNCS gives this prompt by calling `TNC_IMV_SolicitRecommendation`. The IMVs provide their recommendations by calling `TNC_TNCS_ProvideRecommendation`.**
14. The TNCS considers the IMV Action Recommendations supplied by the IMVs and uses an integrity check combining policy to decide what its TNCS Action Recommendation should be.
15. The TNCS sends a copy of its TNCS Action Recommendation to the TNCC. The TNCS also informs the IMVs of its TNCS Action Recommendation via IF-IMV. **[IF-IMV] The TNCS calls `TNC_IMV_NotifyConnectionChange` with the `newState` parameter set to `TNC_CONNECTION_STATE_ACCESS_ALLOWED`, `TNC_CONNECTION_STATE_ACCESS_ISOLATED`, or `TNC_CONNECTION_STATE_ACCESS_NONE`.**

16. The TNCS sends its TNCS Action Recommendation to the NAA. The NAA may ignore or modify this recommendation based on its policies but will typically abide by it.
17. The NAA sends its network access decision to the PEP.
18. The PEP implements the network access decision. During this process, the NAR may be informed of the decision. The TNCC may be informed by the NAR or may discover that a new network has come up.
19. If step 6 was not executed, the network connect process is complete. Otherwise, the TNCC informs the IMCs of the TNCS Action Recommendation via IF-IMC.
20. If the IMCs need to perform remediation, they perform that remediation. Then they continue with Handshake Retry After Remediation. If no remediation was needed, the use case ends here.

7.5 Handshake Retry After Remediation

1. When an IMC completes remediation, it informs the TNCC that its remediation is complete and requests a retry of the Integrity Check Handshake through IF-IMC.
2. The TNCC decides whether to initiate an Integrity Check Handshake retry (possibly depending on policy, user interaction, etc.). Depending on limitations of the NAR, the TNCC may need to disconnect from the network and reconnect to retry the Integrity Check Handshake. In that case (especially if the previous handshake resulted in full access), it may decide to skip the handshake retry. However, in many cases the TNCC will be able to retry the handshake without disrupting network access. It may even be able to retain the state established in the earlier handshake. If the TNCC decides to skip the retry, the use case ends here.
3. The TNCC initiates a retry of the handshake. Skip to step 1, 3, or 5 of the Network Connect section above, depending on which steps are needed to initiate the retry.

7.6 Handshake Retry Initiated by TNCS

1. The TNCS can recheck the security state of the AR periodically or when integrity policies change (such as when a new patch is required) by requesting another Integrity Check Handshake with the TNCC. The handshake retry can be done through the PEP or by communicating directly with the TNCC. State from the previous handshake may be retained or not. An IMV can also request an integrity handshake retry through IF-IMV. If the TNCS decides to skip the Integrity Check Handshake retry, the use case ends here. **[IF-IMV] An IMV requests a handshake retry by calling** `TNC_TNCS_RequestHandshakeRetry`**. The TNCS makes the ultimate decision about whether to retry the handshake. As noted above, the handshake retry may disrupt network connectivity so the TNCS may decide to skip it. In that case, the use case ends here.**
2. The TNCS initiates a retry of the handshake. Skip to step 3 or 5 of the Network Connect section above, depending on whether user authentication will be done in the retry.

7.7 Handshake Retry Where TNCS Goes First

If the underlying protocols (IF-T and IF-TNCCS) support it, the TNCS may send the first batch of messages in a handshake. One common use case where this may be especially useful is with a handshake retry initiated by the TNCS. In this use case, the IMVs can use the first batch of messages to query the IMCs for the information desired. This walkthrough illustrates that use case.

1. The TNCS decides to recheck the security state of the AR for some reason (policy change, periodic recheck, suspicious event, or whatever). The TNCS uses IF-IMV to inform each IMV that an Integrity Check Handshake has started. **[IF-IMV] If this is a new network**

connection, the TNCS calls `TNC_IMV_NotifyConnectionChange` with the `newState` parameter set to `TNC_CONNECTION_STATE_CREATE` to indicate that a new network connection has been created. Then the TNCS calls `TNC_IMV_NotifyConnectionChange` with the `newState` parameter set to `TNC_CONNECTION_STATE_HANDSHAKE`.

2. The TNCS collects the first batch of messages from IMVs that support having the TNCS go first. **[IF-IMV] For IMVs that support having the TNCS go first, the TNCS calls `TNC_IMV_BeginHandshake` as indicated by implementing `"TNC_IMV_BeginHandshake"` function. The IMVs may call `TNC_TNCS_SendMessage` before returning from `TNC_IMV_BeginHandshake` if they want to send instructions to IMC before the IMC can send useful integrity messages. For IMVs that do not support having the TNCS go first, the TNCS does not call `TNC_IMV_BeginHandshake`.**
3. The TNCS passes the IMV messages to the TNCC. This and all other TNCC-TNCS communications can be sent directly but they will often be relayed through one or more of the NAR, PEP, and NAA.
4. The TNCC uses IF-IMC to deliver IMV messages to the IMCs and collects IMC messages that need to be sent back to IMVs in response.
5. Skip to step 8 of the Network Connect section above (section 7.4).

7.8 C Binding Sequence Diagrams

7.8.1 Sequence Diagram for Network Connect

The following sequence diagram (Figure 2) illustrates the Network Connect use case, as described in section 7.4.

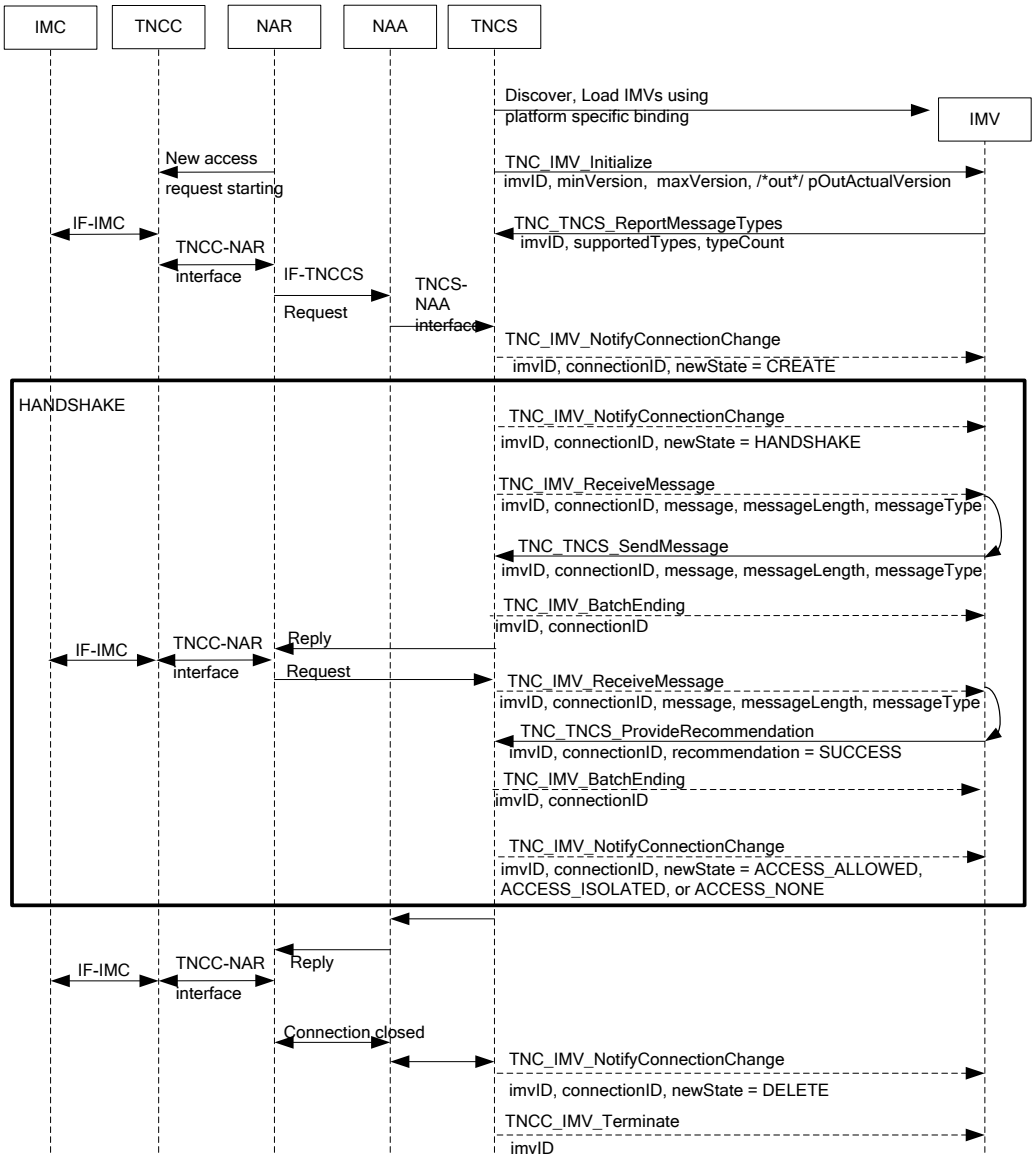


Figure 2 – C Binding: IF-IMV Network Connect Sequence Diagram

7.8.2 Sequence Diagram for Handshake Retry After Remediation

The following sequence diagram (Figure 3) illustrates the Handshake Retry After Remediation use case, as described in section 7.5.

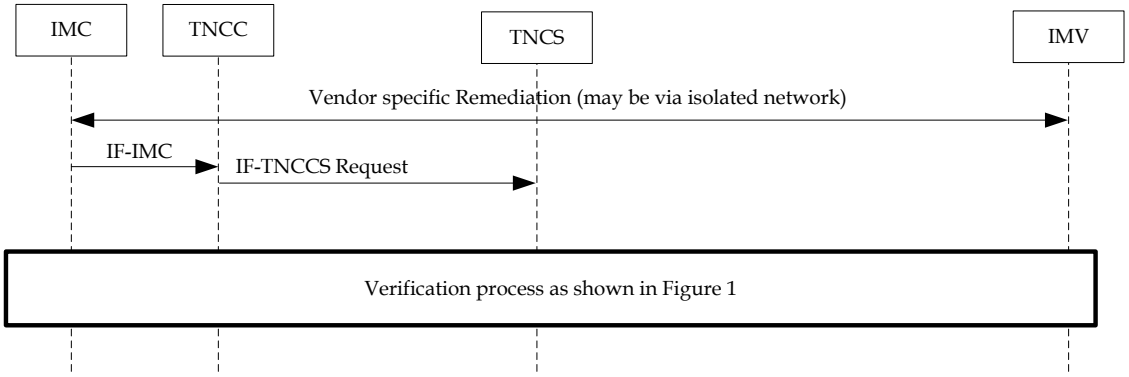


Figure 3 – C Binding: IF-IMV Handshake Retry After Remediation Sequence Diagram

7.8.3 Sequence Diagram for Handshake Retry Initiated by TNCS

The following sequence diagram (Figure 4) illustrates the Handshake Retry Initiated by TNCS use case, as described in section 7.6.

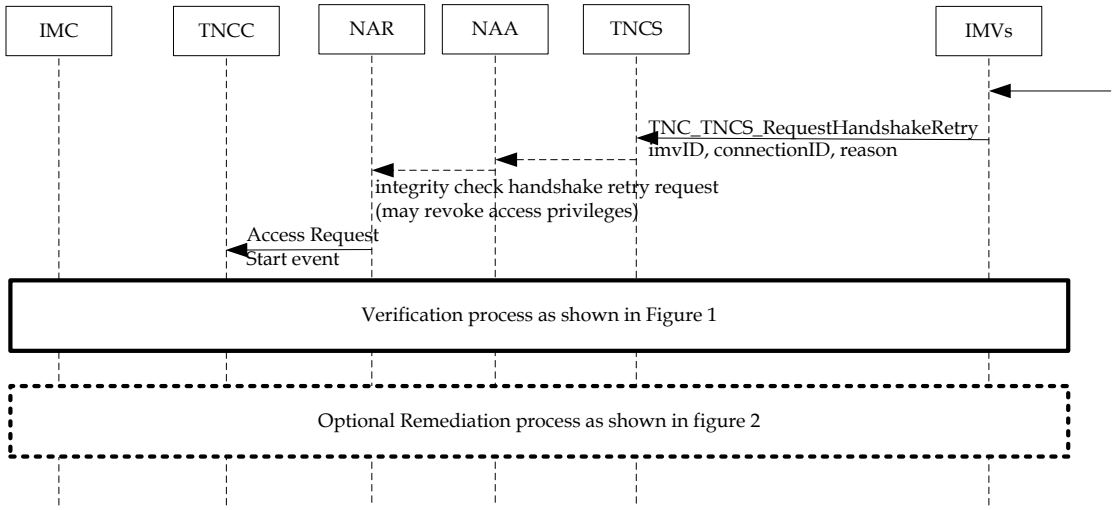


Figure 4 – C Bindng: IF-IMV Handshake Retry Initiated by TNCS

7.8.4 Sequence Diagram for Handshake Retry Where TNCS Goes First

The following sequence diagram (Figure 5) illustrates the Handshake Retry Where TNCS Goes First use case, as described in section 7.7.

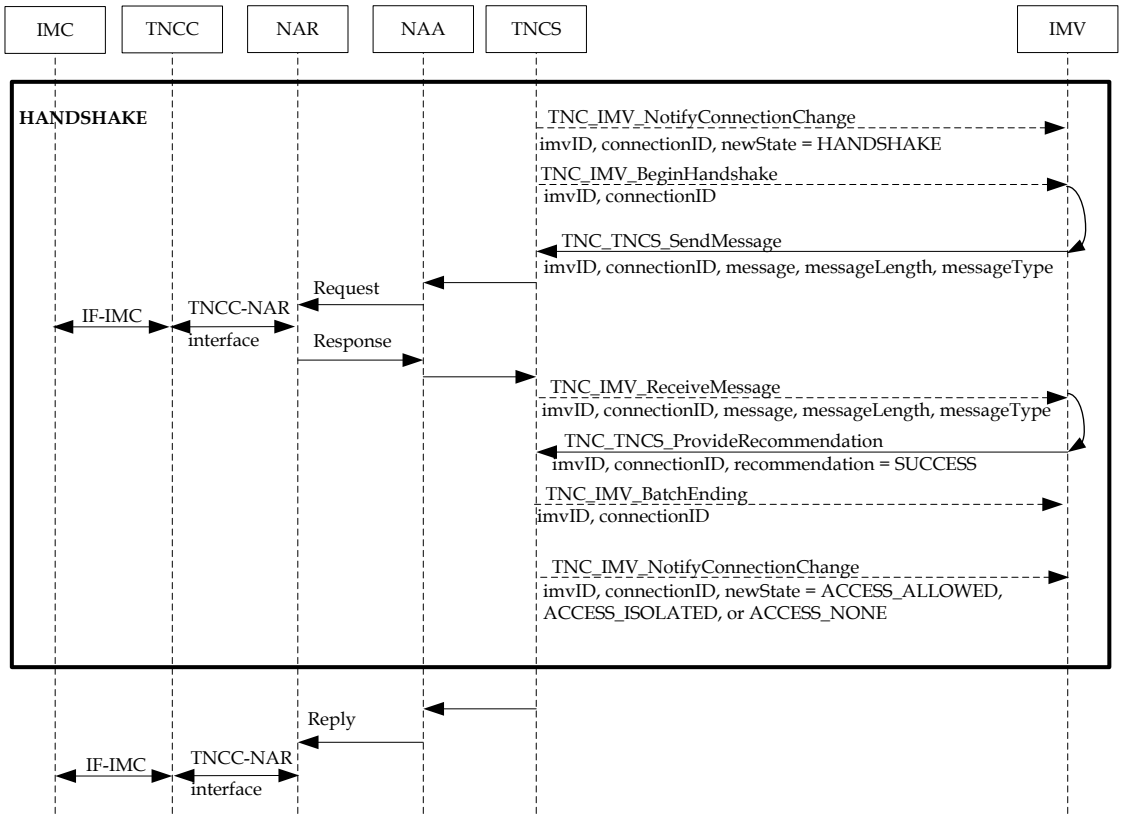


Figure 5 – C Binding: IF-IMV Handshake Retry Where TNCS Goes First Sequence Diagram

7.9 Java Binding Sequence Diagrams

7.9.1 Sequence Diagram for Network Connect

The following sequence diagram (Figure 6) illustrates the Network Connect use case, as described in section 7.4.

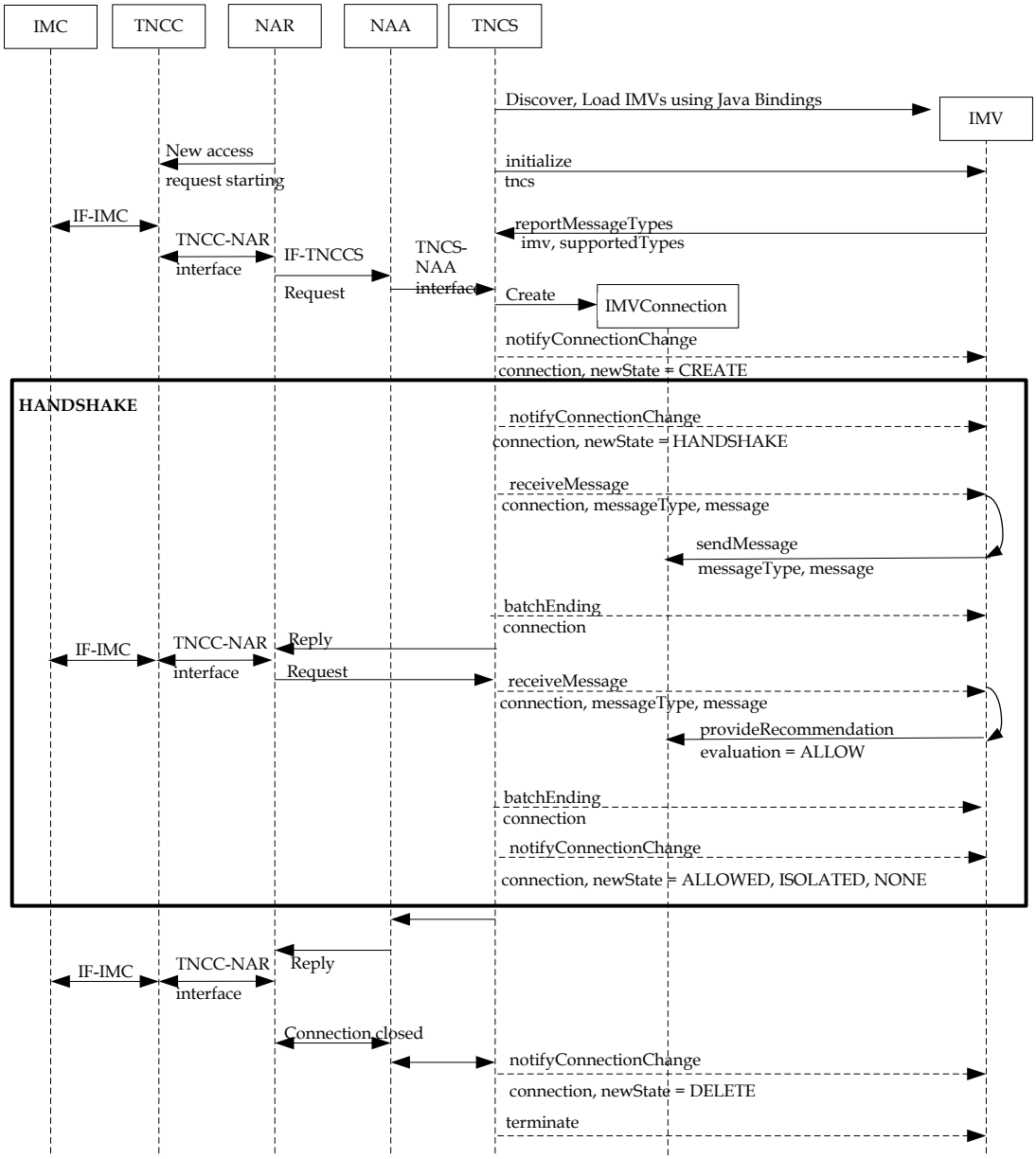


Figure 6 – Java Bindings: IF-IMV Network Connect Sequence Diagram

7.9.2 Sequence Diagram for Handshake Retry After Remediation

The following sequence diagram (Figure 7) illustrates the Handshake Retry After Remediation use case, as described in section 7.5.

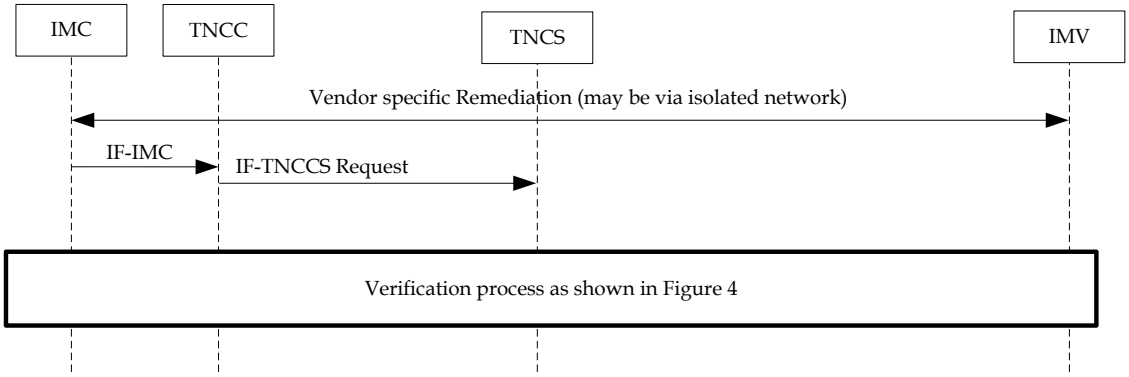


Figure 7– Java Bindings: IF-IMV Handshake Retry After Remediation Sequence Diagram

7.9.3 Sequence Diagram for Handshake Retry Initiated by TNCS

The following sequence diagram (Figure 8) illustrates the Handshake Retry Initiated by TNCS use case, as described in section 7.6.

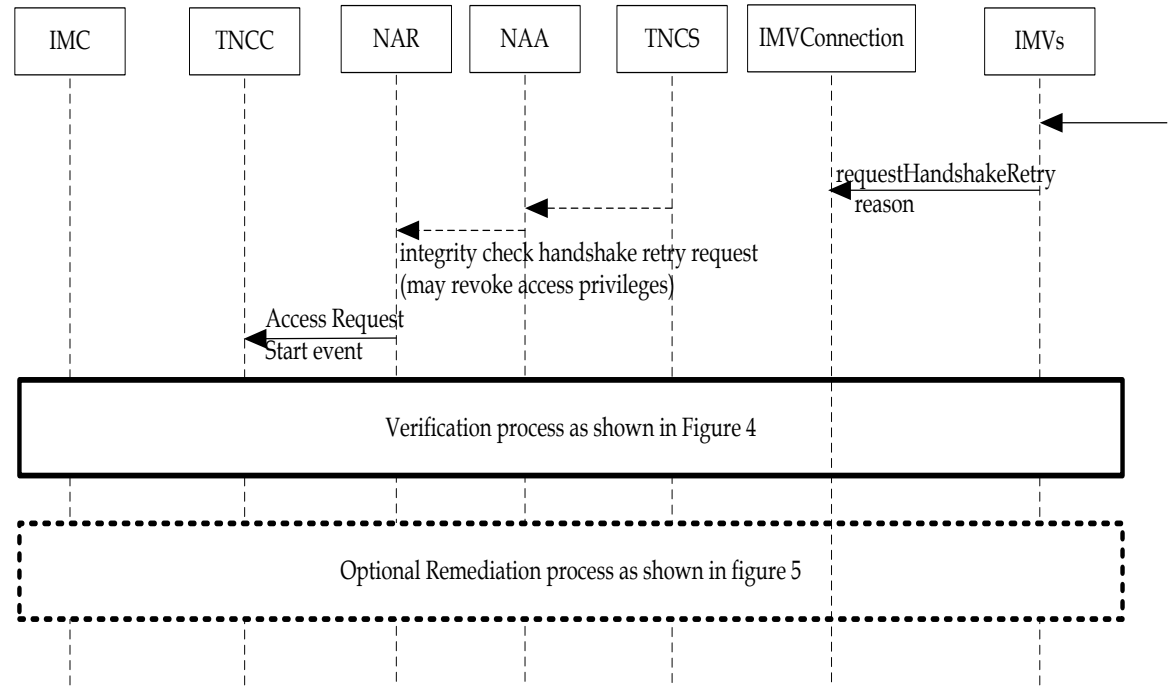


Figure 8 – Java Bindings: IF-IMV Handshake Retry Initiated by TNCS

7.9.4 Sequence Diagram for Handshake Retry With TNCS First

The following sequence diagram (Figure 9) illustrates the Handshake Retry With TNCS First use case, as described in section 7.7.

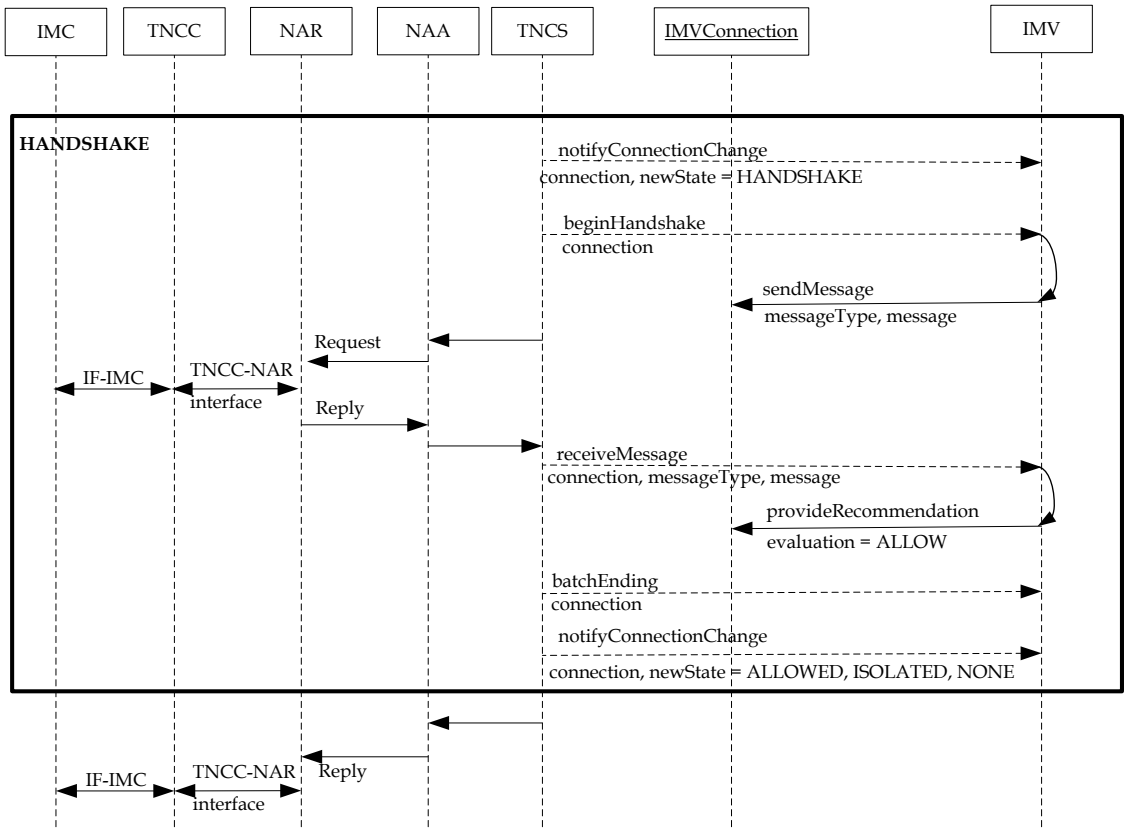


Figure 9 – Java Binding: IF-IMV Handshake Retry With TNCS First Sequence Diagram

8 Implementing a Simple IMV

This section provides a brief informative (non-binding) description of how to implement a simple IMV, one that only checks an IMC's integrity report against a policy and decides based on a recommendation based on this.

This example assumes that you're using the Microsoft Windows DLL platform binding. If not, replace the instructions in section 8.3 about `TNC_IMV_ProvideBindFunction` with your platform's Dynamic Function Binding mechanism.

8.1 Decide on a Message Type and Format

First, you must decide what message type you will use to receive your value from the IMC and what the format of the message will be. This may involve getting a Vendor ID as described in section 3.2.3. Then implement the following functions as described here.

8.2 TNC_IMV_Initialize

All IMVs must implement the `TNC_IMV_Initialize` function. In your implementation, determine whether you support any of the listed IF-IMV API versions. If not, return `TNC_RESULT_NO_COMMON_VERSION`. If so, store the mutually agreed upon version number at `pOutActualVersion` and initialize the IMV. Return `TNC_RESULT_SUCCESS` if all goes well and `TNC_RESULT_FATAL` otherwise. Normally, you might store your IMV ID for later use but in this example all of your code is called by the TNCC so you have the IMV ID as a parameter to all your functions.

8.3 TNC_IMV_ProvideBindFunction

Use the bind function to get a pointer to `TNC_TNCS_ReportMessageTypes`. Then use this pointer to call `TNC_TNCS_ReportMessageTypes` and report which message types you want to receive. Also use the bind function to get a pointer to `TNC_TNCS_ProvideRecommendation` for later use. This is the only state you need to keep. Return `TNC_RESULT_SUCCESS` unless an error occurs. In that case, return `TNC_RESULT_FATAL`.

8.4 TNC_IMV_ReceiveMessage

When you receive a message from an IMC, evaluate it against your policy and then report your recommendation by calling the `TNC_TNCS_ProvideRecommendation` function using the pointer that you saved earlier. If `TNC_TNCS_ProvideRecommendation` returns an error, then return that. Otherwise, return `TNC_RESULT_SUCCESS`.

8.5 TNC_IMV_SolicitRecommendation

If you never received a message from an IMC, you will be prompted to supply a recommendation by a call to `TNC_IMV_SolicitRecommendation`. Probably you will want to recommend against network access since your IMC is not loaded. In any case, report your recommendation by calling the `TNC_TNCS_ProvideRecommendation` function using the pointer that you saved earlier. If `TNC_TNCS_ProvideRecommendation` returns an error, then return that. Otherwise, return `TNC_RESULT_SUCCESS`.

8.6 All Done!

That's it! You've implemented your first IMV. If you need to do anything special on termination, you can implement `TNC_IMV_Terminate`. But many IMVs won't need to.

9 References

9.1 Normative References

[1] Trusted Computing Group, *TNC Architecture for Interoperability*, Specification Version 1.5, May 2012.

[2] Bradner, S., “Key words for use in RFCs to Indicate Requirement Levels”, Internet Engineering Task Force RFC 2119, March 1997.

[3] Alvestrand, H., “Content Language Headers”, Internet Engineering Task Force RFC 3282, May 2002.

[4] Crocker, D., P. Overell, “Augmented BNF for Syntax Specifications: ABNF”, Internet Engineering Task Force RFC 5234, January 2008.

[5] Klensin, J., “Internationalized Domain Names for Applications (IDNA): Definitions and Document Framework”, Internet Engineering Task Force RFC 5890, August 2010.

[6] Klensin, J, Y. Ko, “Overview and Framework for Internationalized Email”, Internet Engineering Task Force RFC 6530, February 2012.

[7] Wahle, M., S. Kille, T. Howes, “Lightweight Directory Access Protocol (v3): UTF-8 String Representation of Distinguished Names”, Internet Engineering Task Force RFC 2253, December 1997.

9.2 Informative References

[8] ISO, ISO/IEC 9899:1999, Programming Languages – C, 1999.

[9] Trusted Computing Group, *TNC IF-IMC*, Specification Version 1.3, February 2013.

[10] Trusted Computing Group, *TNC IF-T: Protocol Bindings for Tunneled EAP Methods,* Specification v1.0, May 2006.

[11] Trusted Computing Group, *TNC IF-T: Protocol Bindings for Tunneled EAP Methods,* Specification v1.1, May 2007.

[12] Trusted Computing Group, *TNC IF-T: Protocol Bindings for Tunneled EAP Methods,* Specification v2.0, TBD.

[13] Trusted Computing Group, *TNC IF-TNCCS-SOH*, Specification Version 1.0, May 2007.

[14] Trusted Computing Group, *TNC IF-TNCCS: TLV Binding*, Specification Version 2.0, January 2010.

[15] Trusted Computing Group, *TNC IF-TNCCS*, Specification Version 1.0, May 2006.

[16] Trusted Computing Group, *TNC IF-TNCCS*, Specification Version 1.1, February 2007.

[17] Microsoft Corporation, *[MS-PEAP]: Protected Extensible Authentication Protocol (PEAP) Specification*, Version 3.1, March 14, 2008.

[18] Trusted Computing Group, *TCG Attestation PTS Protocol: Binding to TNC IF-M,* Specification v1.0, September 2011.

[19] Trusted Computing Group, *TNC IF-T: Binding to TLS,* Revision 2.0, February 2013.

[20] Trusted Computing Group, *IF-M: TLV Binding*, Specification Version 1.0, March 2010.

[21] Sahita, R., Hanna, S., Hurst, R., and K. Narayan, "PB-TNC: A Posture Broker (PB) Protocol Compatible with Trusted Network Communications (TNC)", RFC 5793, March 2010.

[22] Sangster, P., Cam-Winget, N., and J. Salowey, “A Posture Transport Protocol over TLS (PT-TLS)”, RFC 6876, February 2013.

[23] Trusted Computing Group, *TNC IF-IMV: Protocol Binding*, Specification Version 1.0, Work In Progress.

[24] Trusted Computing Group, *TNC IF-T: Binding to TLS,* Specification Version 2.0, February 2013.