TCG Trusted Network Connect
TNC IF-IMC

Specification Version 1.3
Revision 18
27 February 2013
Published

Contact:
admin@trustedcomputinggroup.org

TCG PUBLISHED

Copyright © TCG 2005-2013

TCG

Copyright © 2005-2013 Trusted Computing Group, Incorporated.

Disclaimer

THIS SPECIFICATION IS PROVIDED "AS IS" WITH NO WARRANTIES WHATSOEVER, INCLUDING ANY WARRANTY OF MERCHANTABILITY, NONINFRINGEMENT, FITNESS FOR ANY PARTICULAR PURPOSE, OR ANY WARRANTY OTHERWISE ARISING OUT OF ANY PROPOSAL, SPECIFICATION OR SAMPLE. Without limitation, TCG disclaims all liability, including liability for infringement of any proprietary rights, relating to use of information in this specification and to the implementation of this specification, and TCG disclaims all liability for cost of procurement of substitute goods or services, lost profits, loss of use, loss of data or any incidental, consequential, direct, indirect, or special damages, whether under contract, tort, warranty or otherwise, arising in any way out of use or reliance upon this specification or any information herein.

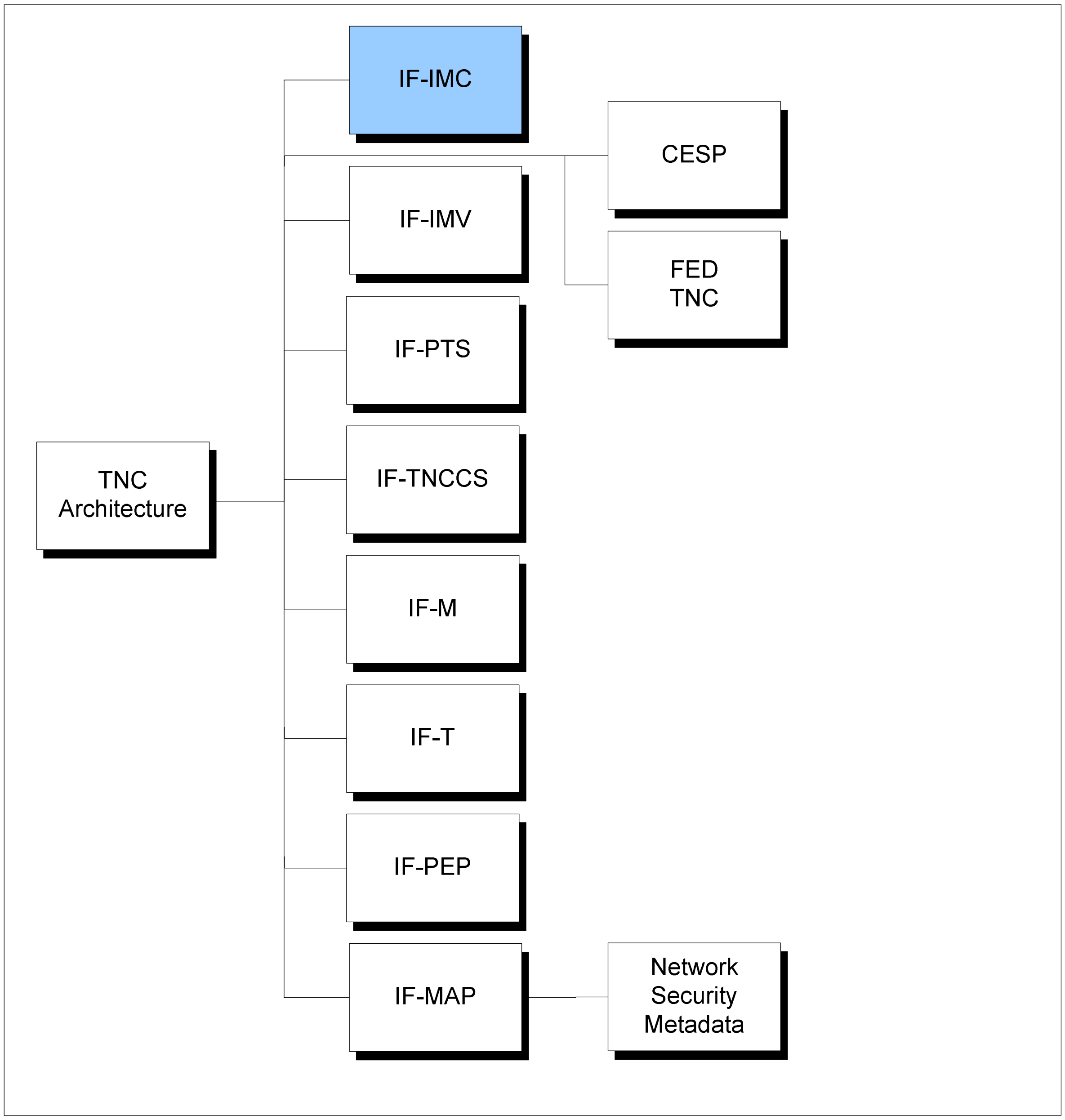
No license, express or implied, by estoppel or otherwise, to any TCG or TCG member intellectual property rights is granted herein.

Except that a license is hereby granted by TCG to copy and reproduce this specification for internal use only.

Contact the Trusted Computing Group at www.trustedcomputinggroup.org for information on specification licensing through membership agreements.

Any marks and brands contained herein are the property of their respective owners.

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 TCG participants contributing to this document:

Aman Garg	3Com
Bipin Mistry	3Com
David Louin	AMOSSYS
Scott Kelly	Aruba
Mahalingam Mani	Avaya
Padma Krishnaswamy	Battelle Memorial Institute
Jeff Dion	Boeing
Steve Venema	Boeing
Nancy Cam-Winget	Cisco Systems
Scott Pope	Cisco Systems
Mark Beadles	Endforce, Inc.
Mark Townsend	Enterasys
Michael McDaniels	Extreme Networks
Ronald Marx	Fraunhofer Institute for Secure Information Technology
Hidenobu Ito	Fujitsu Limited
Sung Lee	Fujitsu Limited
Kazuaki Nimura	Fujitsu Limited
Boris Balacheff	Hewlett-Packard
Paul Crandell	Hewlett-Packard
Graeme Proudler	Hewlett-Packard
Mauricio Sanchez	Hewlett-Packard
Ira McDonald	High North
Andreas Steffen	Hochschule für Technik Rapperswil
Jiwei Wei	Huawei
Han Yin	Huawei
Diana Arroyo (Editor)	IBM
Guerney Hunt	IBM
Lee Terrell	IBM
Frank Yeh	IBM
Chris Hessing	Identity Engines
Stuart Bailey	Infoblox
Navin Boddu	Infoblox
James Tan	Infoblox
David Vigier	Infoblox
Tina Bird	InfoExpress, Inc.
Kent Landfield	Intel Corporation
Ravi Sahita	Intel Corporation
Ned Smith	Intel Corporation
Barbara Nelson	iPass
Chris Trytten	iPass
Josh Howlett	JANET (UK)
Steve Hanna (Editor, TNC-WG co-chair)	Juniper Networks, Inc.
Clifford Kahn	Juniper Networks, Inc.
PJ Kirner	Juniper Networks, Inc.
Lisa Lorenzin	Juniper Networks, Inc.
Hirendra Rathor (Editor)	Juniper Networks, Inc.

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 Corporation
Sandilya Garimella	Motorola
Joseph Tardo	Nevis Networks
Pasi Eronen	Nokia Corporation
Meenakshi Kaushik	Nortel Networks
Thomas Hardjono	SignaCert, Inc.
Carolin Latze	Swisscom
Babak Salimi	Sygate Technologies, Inc.
Bryan Kingsford	Symantec
Paul Sangster	Symantec
Charles Schmidt	The MITRE Corporation
Curtis Simonson	University of New Hampshire InterOperability Lab
Brad Upson	University of New Hampshire InterOperability Lab
Jessica Fitzgerald-McKay	U.S. National Security Agency
Lauren Giroux	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 Connect 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-IMC, a standard interface between Integrity Measurement Collectors and the TNC Client. This document defines and specifies IF-IMC.

Architects, designers, developers, and technologists who wish to implement, use, or understand IF-IMC 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 Background

2.1 Purpose of IF-IMC

This document describes and specifies IF-IMC, a critical interface in the Trusted Computing Group's Trusted Network Connect (TNC) architecture. IF-IMC is the interface between Integrity Measurement Collectors (IMCs) and a TNC Client (TNCC). It is closely related to IF-IMV [5], the interface between Integrity Measurement Verifiers (IMVs) and a TNC Server (TNCS).

IF-IMC is primarily used by the TNC Client to gather the security state in the form of integrity measurements from IMCs, so they can be communicated to 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. IF-IMC also allows IMCs to coordinate with the TNC Client as needed. A more detailed description of the features provided by the IF-IMC API is provided in section 2.9.

2.2 Summary of Changes since IF-IMC 1.2

The following changes have been made to IF-IMC since the last version (IF-IMC 1.2):

- Added optional support to help IMCs and TNCCs change their behavior to accommodate characteristics of specific IF-TNCCS and IF-T protocols.

2.3 Supported Use Cases

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

- A TNCC and one or more IMCs that support the same platform binding have been installed on an endpoint. The TNCC finds and loads the IMCs. Then it runs one or more Integrity Check Handshakes. The IMCs and TNCC may use any of the features of IF-IMC.
- A TNCC that supports the Java Platform Binding has restricted privileges/permissions (as when loaded into a sandbox in a web browser). It loads IMCs that support the Java Platform Binding and runs one or more Integrity Check Handshakes. The IMCs may actually have greater privileges than the TNCC (or they may not be sandboxed).
- A TNCC that supports the Java Platform Binding runs with generous privileges but chooses to run IMCs with restricted privileges for security reasons. It loads IMCs and runs one or more Integrity Check Handshakes.
- A TNCC is running on an endpoint. When an IMC is installed or uninstalled, the TNCC notices this and loads or unloads the IMC.
- A TNCC provides information to IMCs that they can use to adapt their behavior to accommodate one or more of the following cases:
    - A TNCC provides IMCs information about a limited number of round trips and/or a small amount of data for IMC-IMV messages. IMCs can use this information to adapt their messaging to accommodate these limitations.
    - A TNCC that supports Diffie-Hellman Pre-Negotiation (as described in IF-T for Tunneled EAP Methods [7]) provides a Unique-Value-1 to the PTS-IMC or other IMCs, so that this value can be used in the TPM_Quote operation, as described in [14].
    - A TNCC that supports TLS-Unique (as described in PT-TLS [18]) provides the TLS-Unique value to the PTS-IMC or other IMCs, so that this value can be used in the TPM_Quote operation, as described in [14].

TCG Copyright

- A TNCC that supports IF-TNCCS-SOH 1.0 [9] allows an IMC to use IF-TNCCS-SOH features that would not otherwise be supported by IF-IMC: extra fields in SSoHR and SOHEntries and SOHResponseEntries, etc.
- A TNCC that supports IF-TNCCS 2.0 [10] allows an IMC to use IF-TNCCS 2.0 features that were not supported by previous versions of IF-IMC: TNCS sending the first message in a handshake, IMCs getting TNCS preferred language(s), and IMCs sending and receiving IMC-IMV messages with extended values (8 flag bits, 32 bit subtype, and source and destination IMV ID).
- An IMC can determine which IF-TNCCS and IF-T protocol and version are being used for a particular connection.

2.4 Non-supported Use Cases

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

- A TNCC installs an IMC on an endpoint.

2.5 Requirements

Here are the requirements that IF-IMC must meet in order to successfully play its role in the TNC architecture.

- Meets the needs of the TNC architecture

  The API must support all the functions and use cases described in the TNC architecture as they apply to the relationship between the TNC Client and IMC components. The API must support multiple TNCCs on a single AR and multiple overlapping network connections and Integrity Check Handshakes for a single TNCC. The API must allow an IMC to act as a front end for one or more applications or to handle everything within the IMC, as determined by the IMC implementer.

- 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.

  The security requirements for IF-IMC include requirements that unauthorized parties cannot observe communications between the IMC and the TNC Client and that only authorized IMCs can communicate with the TNC Client across IF-IMC. See the Security Considerations section of this document for detailed discussion.

- 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 is also important when you consider that some network endpoints are small and low powered.

- Extensible

IF-IMC needs to expand over time as new features are added to the TNC architecture. The IF-IMC API 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.

- Easy to use and implement

  IF-IMC should be easy for TNC Client and IMC 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. The API should also make things easy for system administrators and end-users. Components of the TNC architecture should plug together automatically without requiring manual configuration.

- Platform-independent

  Since most or all endpoints on a network will be subject to integrity checks, the IF-IMC API must function on as many 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.

- Language-independent

  The IF-IMC API must support the widest possible variety of 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.

- Allow Java IMCs and TNCCs 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.

2.6 Non-Requirements

Here are certain requirements that IF-IMC explicitly is not required to meet. This list is not exhaustive (complete).

- Supporting communications between IMCs and TNCCs written in different languages

  While the IF-IMC API must support the widest possible variety of languages (C, C++, C#, Java, Visual Basic, etc.), it is not required to provide a standard manner for a TNCC written in one language (like C) to load an IMC written in one language (like Java). Depending on the platform and language, this can be quite difficult. When it’s not difficult, we will support such interoperation (as with the Microsoft Windows DLL Binding, which supports interoperation among all languages that can easily call and/or implement a DLL). But support for such cross-language compatibility is not required. Future versions of this API may add such support (perhaps via an RPC mechanism).

- Supporting several TNCCs from different vendors running in a single Java Virtual Machine at the same time

  The Java Platform Binding for IF-IMC requires the TNCC to supply certain standard classes and interfaces. Having two implementations of these classes in a single Java Virtual Machine would cause problems.

2.7 Assumptions

Here are the assumptions that IF-IMC API 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, consistent with the description of message delivery in section 2.10.4 of this specification. In the event that reliable delivery cannot be provided, the TNC Client or TNC Server is expected to terminate the connection.

2.8 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.

2.9 Abstract API Naming Conventions

To avoid name conflicts, all identifiers in the IF-IMC Abstract API have a name that begins with "`TNC_`". Note that this only pertains to the IF-IMC 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 IMC have a name that begins with "`TNC_IMC_`". 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 Client (known as "callbacks") have a name that begins with "`TNC_TNCC_`". 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.

2.10 Features Provided by IF-IMC

This section documents the features provided by IF-IMC.

2.10.1 Integrity Check Handshake

One of the primary functions of IF-IMC 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 IMV Action Recommendations.

2.10.2 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 TNCC provides connection management features.

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

A TNCC SHOULD maintain the same network connection ID across many 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. Second, it allows an IMC to request a handshake retry for a particular connection, as when the IMC has completed remediation requested by an IMV. A TNCC SHOULD ensure that connection IDs persist long enough to permit handshake retry. Since remediation may require restarting the operating system, power cycling, and other measures, the connection ID SHOULD be remembered even across these measures so that the handshake can be retried after remediation. However, the TNCS MAY refuse to maintain state on old handshakes, forcing a full handshake every time. This is fine. The network connection ID is assigned by the TNCC. Little or no cooperation from the TNCS is required to allow the TNCC to maintain the network connection ID. The TNCC MUST use the same connection ID for all IMCs when referring to a particular connection.

If more than one TNC Client may be running at once on a single machine (rare, but possible) and an IMC is loaded by both TNC Clients, the IMC MUST work properly even if the TNC Clients happen to choose the same network connection ID for different connections. This should not be too hard, since the IMCs will be loaded separately. However, it may become an issue if the IMCs need to communicate with a common application and refer to the network connections.

2.10.3 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 1.0 and IF-T for Tunneled EAP Methods do not provide any support for handshake retry without disrupting network connectivity, but IF-TNCCS 2.0 and IF-T for TLS do.

To support handshake retries, the TNCC SHOULD maintain a network connection ID after an Integrity Check Handshake has been completed. This network connection ID can then be used by the TNCC to inform IMCs that it is retrying the handshake or by an IMC to request a retry (due to remediation 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-IMC supports handshake retry and requires IMCs to handle handshake retries (usually trivial) but does not require TNCCs to honor IMC requests for handshake retry. In fact, IF-IMC requires an

IMC to provide information about the reason for requesting handshake retry so that the TNCC or 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. There is no special support in IF-IMC for this feature. If remediation instructions require network access, IMCs SHOULD NOT follow them until the network connection state changes to success or isolated.

Incompatible IMV policies (more likely with multiple network connections) can cause flip-flopping or other problems if an IMC receives conflicting remediation instructions from different IMVs. IMCs may want to detect this as much as possible and notify the user or administrator or ask them for guidance or refuse to perform remediation that would cause them to flip-flop.

2.10.4 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 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 in section 3.5.2.5. From the perspective of IF-IMC and the TNCC and TNCS, this method is not important. The message type is simply a number.

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, so this flag does not appear in IF-IMC. All messages sent by an IMC through IF-IMC have a recipient type of IMV. All messages received by an IMC through IF-IMC have a recipient type of IMC. 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-IMC allows an IMC 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 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 any appropriate protocol.

All messages sent with `TNC_TNCC_SendMessage` and received with `TNC_IMC_ReceiveMessage` are between the IMC and IMV. The IMC communicates with the

TNCC by calling functions (standard and vendor-specific) in IF-IMC, not by sending messages. The TNCC should not interfere with communications between the IMC and IMVs by consuming or blocking IMC-IMV messages.

2.10.5 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 always groups 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.

To clarify which IMV functions the TNCC calls under which circumstances, please consult the sequence diagrams in section 7.

An IMC can only send a message in three circumstances: during the initial batch (when `TNC_IMC_BeginHandshake` is called), in response to a message received by the IMC in a later batch (when `TNC_IMC_ReceiveMessage`, `TNC_IMC_ReceiveMessageSOH`, or `TNC_IMC_ReceiveMessageLong` is called), and at the end of a batch (when `TNC_IMC_BatchEnding` is called). At any of these times, the IMC MAY send one or more messages by calling `TNC_TNCC_SendMessage`, `TNC_TNCC_SendMessageSOH`, or `TNC_TNCC_SendMessageLong` once for each message to be sent and then returning from `TNC_IMC_BeginHandshake`, `TNC_IMC_ReceiveMessage`, `TNC_IMC_ReceiveMessageSOH`, `TNC_IMC_ReceiveMessageLong`, or `TNC_IMC_BatchEnding`. Note that if the IMC does not call `TNC_TNCC_SendMessage`, `TNC_TNCC_SendMessageSOH`, or `TNC_TNCC_SendMessageLong` before returning from `TNC_IMC_BeginHandshake`, `TNC_IMC_ReceiveMessage`, `TNC_IMC_ReceiveMessageSOH`, `TNC_IMC_ReceiveMessageLong`, or `TNC_IMC_BatchEnding`, this indicates that it does not want to send any messages at this time. IMVs 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 IMC that is not engaged in a dialog with an IMV may well find that the handshake has ended.

When an Integrity Check Handshake is beginning and the TNCC wants to solicit messages from IMCs for the first batch, it calls `TNC_IMC_BeginHandshake` for each IMC. This indicates to the IMCs that an Integrity Check Handshake is beginning and they should send any IMC-IMV messages. IMCs send those messages by calling the `TNC_TNCC_SendMessage` function before returning from `TNC_IMC_BeginHandshake`. Once all IMCs have finished sending their messages for a batch, the TNCC will send those messages to the TNCS and await its response. When this response is received, the TNCC will deliver to IMCs any messages sent by IMVs and start accepting messages from IMCs.

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

As with all IMC functions, the IMC SHOULD NOT wait a long time (definitely not more than a second) before returning from `TNC_IMC_BeginHandshake`, `TNC_IMC_ReceiveMessage`, `TNC_IMC_ReceiveMessageSOH`, or `TNC_IMC_ReceiveMessageLong`, or `TNC_IMC_BatchEnding`. A long delay might frustrate users or exceed network timeouts (PDP, PEP, or otherwise). IMCs that need to perform a lengthy process may want to simply send a status message, indicating that they are working. The IMVs can respond in the next batch with a status query, and thus the handshake can be kept 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 TNCC will return `TNC_RESULT_ILLEGAL_OPERATION` from `TNC_TNCC_SendMessage`, `TNC_TNCC_SendMessageSOH`, or `TNC_TNCC_SendMessageLong`. If the TNCC supports limiting the message size or number of round trips, the TNCC MUST return `TNC_RESULT_EXCEEDED_MAX_MESSAGE_SIZE` or `TNC_RESULT_EXCEEDED_MAX_ROUND_TRIPS` respectively if the limits are exceeded. An IMC can get the Maximum Message Size and Maximum Number of Round Trips by using the related Attribute ID within the TNC_TNCC_GetAttribute function. The IMC SHOULD adapt its behavior to accommodate these limitations if available.

3 IF-IMC Abstract API

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

3.1 Platform and Language Independence

IF-IMC 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 [6]) 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 IMCs and TNCCs 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 IMCs and TNC Clients on their platform. This specification does not provide a standard way to mix an IMC written in one language with a TNCC written in another language, beyond the support that may be provided by platform-specific bindings.

IF-IMC is also a platform-independent API. It is designed to support almost any platform. Platform-specific bindings are described in section 3.9.5. The IF-IMC 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-IMC 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-IMC API. Future versions may be incompatible due to removing, adding, or changing functions, types, and constants. However, the `TNC_IMC_Initialize` function and its associated types and constants will not change, so that version incompatibilities can be detected. A TNCC or IMC can even support multiple versions of the IF-IMC API for maximum compatibility. See section 3.8.1 for details.

3.2.2 Dynamic Function Binding

Platforms that support IF-IMC SHOULD support dynamic function binding. This feature allows a TNCC or IMC 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 TNCC or IMC 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 TNCCs and IMCs that support different version numbers interact safely.

3.2.3 Vendor IDs

The IF-IMC API supports several forms of vendor extensions. IMC or TNCC vendors can define vendor-specific functions and make them available to the other party. IMC or TNCC vendors can define vendor-specific result codes and vendor-specific attribute IDs. And IMC 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-IMC 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, 3.5.2.12, and 3.5.2.5.

3.2.4 Vendor-Specific Functions

The IMC and TNC client MAY extend the IF-IMC API by defining vendor-specific functions that go beyond those described here. An IMC or TNC Client 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_IMC_Reinstall`" is clearly intended to be implemented by IMCs.

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-IMC now includes several functions and features that allow IMCs to adapt their behavior to underlying protocols. This section enumerates those features. It also describes how TNCCs are expected to behave with each of the underlying protocols, since their behavior must be predictable to ensure interoperability.

3.3.1 IMC Adaptation to Lower Layer Protocols

Simple IMCs may not need to adapt their behavior to differences in lower layer protocols. For example, an IMC that just sends a short message when a handshake begins should not need to adapt. However, an IMC that wants to engage in a multi-round handshake or send 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, a chatty IMC may just send a short summary report instead.

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

- An IMC 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 IMC to adapt its behavior in a generic manner to whatever the underlying protocols may be.
- An IMC can use protocol-specific functions or attributes to take advantage of features that are specific to a particular protocol. For example, the `TNC_TNCC_SendMessageSOH` function allows an IMC to send fields that are specific to the IF-TNCCS-SOH protocol.
- An IMC can read protocol type and version attributes and adapt its behavior to one specific protocol version. For example, an IMC that detects the IF-TNCCS 2.0 protocol may send messages in the IF-M 1.0 format since those protocols are generally used together.

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

3.3.2 TNCC Behavior With Different Lower Layer Protocols

TNCCs need to change their behavior when different versions of IF-TNCCS are used. For example, a TNCC 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-IMC. Further requirements are contained in the rest of this specification (e.g. in the description of functions like `TNC_IMC_ReceiveMessageSOH`).

3.3.2.1 TNCC 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 IMCs so that they can adapt and enforce those limitations if the IMCs do not adapt. A TNCC that implements IF-TNCCS-SOH and allows IMCs 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 TNCC to ensure that if each of the IMCs complies with this constraint then the total size of the SoH message will not exceed 4000 bytes (or whatever the maximum size for the SoH message is for this session). 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 SoH header, SSoH, etc.) from 4000 and divide by the number of IMCs. A more sophisticated method might provide lower values to IMCs that generally only send short messages, thereby allowing other IMCs 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 IMCs can use to access these fields: `TNC_IMC_ReceiveMessageSOH`, `TNC_TNCC_SendMessageSOH`, and the Has SOH and SSoHR attributes. A TNCC that allows IMCs 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 TNCC 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 TNCC 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 TNCC's behavior proceeds as usual: call the `TNC_IMC_NotifyConnectionChange` function for IMCs that are participating in the handshake with the value `TNC_CONNECTION_STATE_HANDSHAKE`, call the `TNC_IMC_BeginHandshake` function, and see what messages the IMCs send. One difference with IF-TNCCS-SOH is that the TNCC SHOULD allow the IMC to get the Maximum Round Trips, Maximum Message Size, and Has SOH attributes. Also, the TNCC SHOULD implement the `TNC_TNCC_SendMessageSOH` function so that the IMC can send full SOHReportEntry messages. Finally, the TNCC MUST modify the behavior of the `TNC_TNCC_SendMessage` function so that each message sent by an IMC using this function is stuffed into a Vendor-Specific atrribute in an SOHReportEntry message, as described in the description of the `TNC_TNCC_SendMessage` function. When all the IMCs' messages have been sent (generally when all IMCs have returned from `TNC_IMC_BeginHandshake`), the TNCC should send the SOH message.

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

When a TNCC receives an SoHR message, it should prepare the message so that IMCs can access it with the SOHR and SSoHR attributes. After that, the TNCC should parse the SOHR message into SOHResponseEntry 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 IMCs' most recent calls to `TNC_TNCC_ReportMessageTypes` or `TNC_TNCC_ReportMessageTypesLong`. The SOHResponseEntry should be delivered to each IMC that has a match. If an IMC does not support `TNC_IMC_ReceiveMessageSOH` (or when the TNCC does not support that function), the TNCC 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_IMC_ReceiveMessage`. If an IMC does support `TNC_IMC_ReceiveMessageSOH`, the TNCC should deliver the entire SOHRReportEntry using that function. Since there is only one round trip in IF-TNCCS-SOH, the handshake is completed once the SOHRReportEntries have been delivered. The TNCC should then call `TNC_IMC_BatchEnding` to indicate that the batch is ending and `TNC_IMC_NotifyConnectionChange` to indicate what the result of the handshake was. The result of the handshake can be determined by looking at the `qState` field in the `MS-Quarantine-State` attribute in the SSoHR. If this value is 1 or 2, the new connection state is `TNC_CONNECTION_STATE_ACCESS_ALLOWED`. If the value is 3, the new connection state is `TNC_CONNECTION_STATE_ACCESS_ISOLATED`.

3.3.2.2 TNCC 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 TNCC that supports the IF-TNCCS 2.0 protocol should implement the IF-IMC interface so that IMCs know what to expect.

A TNCC 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 TNCC (i.e. a TNCC that implements the IF-TNCCS 2.0 protocol) MUST also implement the `TNC_TNCC_SendMessageLong` and `TNC_TNCC_ReserveAdditionalIMCID` functions and call the `TNC_IMC_ReceiveMessageLong` function as described in the description of that function. And such a TNCC 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 TNCC that implements the IF-TNCCS 2.0 protocol MUST implement the IMC Supports TNCS First attribute. If the TNCS sends the first batch in an IF-TNCCS 2.0 handshake, the TNCC MUST deliver messages contained in this batch to IMCs that have set a value of 1 for the IMC Supports TNCS First attribute and MUST NOT deliver those messages to the other IMCs.

When the IF-TNCCS 2.0 protocol is used, the TNCC MUST assign a 16-bit IMC Identifier to each IMC and send this in the IMC Identifier field of the TNCCS-IF-M-Message unless the IMC has reserved a different IMC Identifier with `TNC_TNCC_ReserveAdditionalIMCID` and passed this identifier to the `TNC_TNCC_SendMessageLong` function as the `sourceIMCID`. To simplify things, the TNCC MUST use the same value for this IMC identifier that it uses for the 32-bit IMC ID passed to all IMC functions. If a TNCC 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 TNCC SHOULD NOT deliver this message to any IMC other than the one whose IMC Identifier matches the IMC Identifier field of the TNCCS-IF-M-Message. If there is no such IMC or if that IMC has not reported an interest in the message type of the message, the TNCC should discard the message.

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

3.4 Threading and Reentrancy

Threading is addressed in the platform-specific bindings in section 3.9.5.

The TNCC MUST be reentrant (able to receive and process a function call even when one is already underway) and MUST be thread-safe. IMC DLLs are not required to be reentrant. Therefore, the TNC Client MUST NOT call an IMC DLL from a callback function (like `TNC_TNCC_SendMessage`) and MUST NOT call an IMC DLL from one thread if another thread has an active call into that DLL. However, since more than one TNC Client may be running at once on a single machine (rare, but possible), any IMC DLL MUST be prepared to be loaded in multiple processes at once and to have these processes issue overlapping calls to the DLL. An IMC DLL MAY return `TNC_RESULT_CANT_RESPOND` from any function if it is temporarily unable to respond (perhaps because it can only handle one network connection at once). If at all possible, the IMC DLL should avoid doing this since it may cause the TNCC to proceed without the IMC’s messages, resulting in denied network access or even unnecessary remediation.

Note that an IMC DLL may just be a stub that communicates with a separate process that processes and responds to IMV messages. This will be fairly common, especially when there is already a background process running (to do real-time virus checking, for instance). Alternatively, the IMC DLL may be very simple, reporting stored values. This will also be very common, especially when integrity checks are fairly static. The checks can run periodically and store their results. The IMC DLL can just read and report these stored results.

3.5 Data Types

This section describes the data types defined and used in the abstract IF-IMC API.

3.5.1 Basic Types

These types are the most basic ones used by the IF-IMC API. They are defined in a platform-dependent and language-dependent manner to meet the requirements described in this section. Consult section 3.9.5 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_IMCID`	`TNC_UInt32`	**IMC ID**
`TNC_ConnectionID`	`TNC_UInt32`	**Network Connection ID**
`TNC_ConnectionState`	`TNC_UInt32`	**Network Connection State**
`TNC_RetryReason`	`TNC_UInt32`	**Handshake retry reason**
`TNC_MessageType`	`TNC_UInt32`	**Message type**
`TNC_MessageTypeList`	Platform-specific	**Reference to list of TNC_MessageTypes**
`TNC_VendorID`	`TNC_UInt32`	**Vendor ID**
`TNC_VendorIDList`	Platform-specific	**Reference to list of TNC_VendorIDs**
`TNC_MessageSubtype`	`TNC_UInt32`	**Message subtype**
`TNC_MessageSubtypeList`	Platform-specific	**Reference to list of TNC_MessageSubtypes**
`TNC_Version`	`TNC_UInt32`	**IF-IMC API version number**
`TNC_Result`	`TNC_UInt32`	**Result code**
TNC_AttributeID	**TNC_UInt32**	**Attribute ID**

3.5.2.1 IMC ID

When a TNC Client loads an IMC, it assigns it an IMC ID (represented by the `TNC_IMCID` type). This allows the IMC to identify itself when calling TNCC functions. The IMC ID is a `TNC_UInt32` chosen by the TNCC and passed to the `TNC_IMC_Initialize` function. This IMC ID is referred to as primary IMC ID in the document henceforth. It is valid until the TNCC calls `TNC_IMC_Terminate` for this IMC.

An IMC can also request additional IMC IDs, if the TNCC implements the `TNC_TNCC_ReserveAdditionalIMCID` function. The additional IMC IDs are valid until the TNCC calls `TNC_IMC_Terminate` for this IMC. This is useful when IF-M messages are used, as described in section 3.2 of IF-M 1.0 [16]. Refer to section 3.9.7 for details. Most of the IF-IMC functions implemented by either IMC or TNCC accept IMC ID as a parameter. This parameter MUST be the primary IMC ID of the IMC unless specified otherwise in the function documentation.

There is no internal structure to an IMC ID. The IMC ID `TNC_IMCID_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 TNCC can choose any other value for the IMC ID and the IMC MUST NOT attach any significance to the value chosen. However, the TNCC 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 TNCC may be negotiating with several different TNCSs at once (if the endpoint has several network interfaces that are coming up simultaneously, for instance). Each of these TNCC-TNCS pairs is referred to as a “network connection”.

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

The IMC and TNCC 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 IMC messages are sent to the right TNCS and helps the IMC match up messages from IMVs with any state the IMC may be maintaining from earlier parts of that IMC-IMV conversation (even extending across multiple Integrity Check Handshakes in a single network connection).

The TNCC notifies IMCs of changes in network connection state (handshake success, handshake failure, etc.) by calling the `TNC_IMC_NotifyConnectionChange` function. When a network connection is finished, the TNCC first notifies IMCs of this by calling the `TNC_IMC_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 2.10.3 above, it is sometimes desirable to retry an Integrity Check Handshake (when remediation is complete, for instance). Some TNCCs will not support this but all IMCs MUST do so. To indicate that a network connection retry is beginning, a TNCC notifies the IMCs by calling the `TNC_IMC_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 IMC can ask the TNCC to retry an Integrity Check Handshake by calling the `TNC_TNCC_RequestHandshakeRetry` function. For details on this, see the description of that function in section 3.9.4.

There is no internal structure to a network connection ID. There is one reserved value: `TNC_CONNECTIONID_ANY (0xffffffff)`. The TNCC can choose any other value for a network connection ID that does not conflict with another valid network connection ID for the same TNCC-IMC 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 IMC MUST NOT attach any significance to the value chosen.

3.5.2.3 Network Connection State

The TNCC uses the `TNC_IMC_NotifyConnectionChange` function to notify IMCs of changes in network connection state. The network connection state is represented as a `TNC_UInt32`. The TNCC MUST pass one of the values listed in section 3.6.3. The TNCC MUST NOT use any other network connection state value with this version of the IF-IMC API.

3.5.2.4 Handshake Retry Reason

The IMC can ask the TNCC to retry an Integrity Check Handshake by calling the `TNC_TNCC_RequestHandshakeRetry` function. One of the parameters to that function is a `TNC_RetryReason`. This type is represented as a `TNC_UInt32`. The IMC MUST pass one of the values listed in section 3.6.5. The IMC MUST NOT use any other handshake retry reason value with this version of the IF-IMC API.

3.5.2.5 Message Type

As described in section 2.10.4, 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-IMC (`TNC_IMC_ReceiveMessageLong` and `TNC_TNCC_SendMessageLong`).

Of course, the new expanded format for message types will only work with software and protocols that support it. IMCs that wish to use the expanded format should use dynamic function binding to determine whether the TNCC implements the `TNC_TNCC_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 IMC 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 2.10.4 and 3.9.1).

3.5.2.6 Message Type List

The `TNC_MessageTypeList` type represents a list of message types. Its exact representation is platform-specific, but will typically be a pointer or reference to an array of `TNC_MessageType`s.

3.5.2.7 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. IMCs may request messages with this vendor ID to indicate that they want to receive messages whose message type includes any vendor ID. However, an IMC MUST NOT send messages whose message type includes this reserved value and a TNCC 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.

3.5.2.8 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_VendorID`s.

3.5.2.9 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. IMCs may request messages with this message subtype to indicate that they want to receive messages whose message subtype has any value. However, an IMC MUST NOT send messages whose message subtype includes this reserved value and a TNCC 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 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_MessageSubtype`s.

3.5.2.11 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.

3.5.2.12 Result Code

Each function in the IF-IMC 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 3.6.1) 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.

IMCs and TNCCs 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-IMC API. Any unknown non-zero result code SHOULD be treated as equivalent to `TNC_RESULT_OTHER`.

3.5.2.13 Attribute ID

The `TNC_AttributeID` type identifies a TNC attribute. TNC attributes allow IMCs to set and get attribute values identified by a `TNC_AttributeID` and associated with a TNCC or network connection. For instance, an IMC can get the attribute value with attribute ID `TNC_ATTRIBUTE_MAX_ROUND_TRIPS` associated with a particular connection, which identifies the maximum round trips 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 or 0x005597 are reserved for definition by TCG. Some of these reserved attribute IDs are defined in section 3.6.8 below.

Generally, each TNCC will support only a limited set of attribute IDs. TNCCs MAY support no attribute IDs at all. IMCs MUST be prepared for this.

Note that the Java Platform Binding for IF-IMC 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-IMC API.

3.6.1 Result Code Values

Each function in the IF-IMC 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-IMC API. IMCs and TNCCs 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 IMC function returns `TNC_RESULT_FATAL`, then the IMC has encountered a permanent error. The TNCC SHOULD call `TNC_IMC_Terminate` as soon as possible. The TNCC MAY then try to reinitialize the IMC with `TNC_IMC_Initialize` or try other measures such as unloading and reloading the IMC and then reinitializing it.

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

Result Code	Definition
TNC_RESULT_SUCCESS	**Function completed successfully**
TNC_RESULT_NOT_INITIALIZED	**TNC_IMC_Initialize has not been called**
TNC_RESULT_ALREADY_INITIALIZED	**TNC_IMC_Initialize was called twice without a call to TNC_IMC_Terminate**
TNC_RESULT_NO_COMMON_VERSION	**No common IF-IMC API version between IMC and TNC Client**
TNC_RESULT_CANT_RETRY	**TNCC cannot attempt handshake retry**
TNC_RESULT_WONT_RETRY	**TNCC refuses to attempt handshake retry**
TNC_RESULT_INVALID_PARAMETER	**Function parameter is not valid**
TNC_RESULT_CANT_RESPOND	**IMC cannot respond now**
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**

3.6.2 Version Numbers

As noted in section 3.2.1, this specification defines version 1 of the TNC IF-IMC 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_IFIMC_VERSION_1	**The version of IF-IMC API defined here**

3.6.3 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_TNCC_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**

3.6.4 Network Connection State Values

This is the complete set of permissible values for the `TNC_Connection_State` type in this version of the IF-IMC 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.**

3.6.5 Handshake Retry Reason Values

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

Handshake Retry Reason Value	Definition
TNC_RETRY_REASON_IMC_REMEDIATION_COMPLETE	**IMC has completed remediation**
TNC_RETRY_REASON_IMC_SERIOUS_EVENT	**IMC has detected a serious event and recommends handshake retry even if network connectivity must be interrupted**
TNC_RETRY_REASON_IMC_INFORMATIONAL_EVENT	**IMC has detected an event that it would like to communicate to the IMV. It requests handshake retry but not if network connectivity must be interrupted**
TNC_RETRY_REASON_IMC_PERIODIC	**IMC wishes to conduct a periodic recheck. It recommends handshake retry but not if network connectivity must be interrupted**

3.6.6 Vendor ID Values

These are reserved vendor ID values. Other vendor IDs may be used as described in section 3.5.2.7. 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**

3.6.7 Message Subtype Values

This is a reserved message subtype value. Other message subtypes may be used as described in section 3.5.2.8. Note that message subtypes are assigned by vendors as described in section 3.5.2.5.

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

3.6.8 Attribute ID Values and Value Definitions

As described in section 3.5.2.13, attribute IDs with a vendor ID of zero 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_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 IMCs including PTS-IMC
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_SOHR	0x00559708	Contents of SOHR
TNC_ATTRIBUTEID_SSOHR	0x00559709	Contents of SSOHR
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 IMCs including PTS-IMC
TNC_ATTRIBUTEID_IMC_SPTS_TNCS1	0x0055970F	Flag set by IMC to indicate whether IMC supports TNCS sending first message
TNC_ATTRIBUTEID_PRIMARY_IMC_ID	0x00559711	Primary ID of the IMC

3.6.8.1 Preferred Language Attribute

The Preferred Language attribute indicates which human-readable language(s) are preferred for human-readable strings to be sent over a particular connection or for the TNCC as a whole. An IMC may get the value of the Preferred Language attribute with `TNC_TNCC_GetAttribute` but may not set this value with `TNC_TNCC_SetAttribute`. If the IMC provides a connection ID to `TNC_TNCC_GetAttribute`, the attribute value returned by the TNCC will pertain to that connection. Generally, this will be the preferred language of the TNCS associated with that connection. If the IMC provides `TNC_CONNECTIONID_ANY`, the attribute value will pertain to the TNCC as a whole. A TNCC 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 [4] (US-ASCII only, no control characters allowed). This header lists the languages preferred for human-readable messages. The TNCC 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 IMC must be able to handle this case, which may be common if a TNCC supports this function but the TNCS does not provide language preference information (perhaps because underlying protocols do not support passing this 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.

TNCCs are not required to implement the Preferred Language attribute, but they SHOULD do so if possible, since this feature helps with internationalization. Likewise, IMCs SHOULD make use of this function when available but are not required to do so. Many TNCCs do not support this attribute and many connection protocols also do not support it. IMCs MUST work properly if a TNCC does not support it.

Many IMCs don't send human-readable strings over a connection. If they do send such strings (e.g. log messages), they may not have the ability to adapt their strings to the preferences of the TNCS or its operators. Further, many IF-TNCCS protocols do not have the ability to convey information about TNCS language preferences. Still, it's good to have the ability to provide language preference information to IMCs so that they can use it when appropriate.

3.6.8.2 Maximum Round Trips Attribute

The Maximum Round Trips attribute allows a TNCC to indicate to the IMCs the maximum number of round trips the underlying transport supports. An IMC may get the value of the Maximum Round Trips attribute with `TNC_TNCC_GetAttribute` but may not set this value with

`TNC_TNCC_SetAttribute`. The IMC MUST provide a valid connection ID to `TNC_TNCC_GetAttribute`. The attribute value returned by the TNCC 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 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.

TNCCs are not required to implement the Maximum Round Trips attribute, but they SHOULD do so if possible. Likewise, IMCs SHOULD make use of this attribute when available but are not required to do so. If the Maximum Round Trips is exceeded, the TNCC MUST return the TNC_RESULT_EXCEEDED_MAX_ROUND_TRIPS result code from the `TNC_TNCC_SendMessage`, `TNC_TNCC_SendMessageSOH`, or `TNC_TNCC_SendMessageLong` functions. Many TNCCs do not support this attribute; IMCs MUST work properly if a TNCC does not support it.

3.6.8.3 Maximum Message Size Attribute

The Maximum Message Size attribute allows a TNCC to indicate to the IMCs the maximum message size the underlying transport supports. An IMC may get the value of the Maximum Message Size attribute with `TNC_TNCC_GetAttribute` but may not set this value with `TNC_TNCC_SetAttribute`. The IMC MUST provide a valid connection ID to `TNC_TNCC_GetAttribute`. The attribute value returned by the TNCC 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 IMC should send in a single batch (combining the data in all messages sent by that IMC in that batch). A value of zero 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.

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

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

The DHPN attribute allows a TNCC that supports Diffie-Hellman Pre-Negotiation (as described in IF-T for Tunneled EAP Methods) to provide to a PTS-IMC or other IMCs a Unique-Value-1, so that this value can be used in the TPM_Quote operation. An IMC may get the value of the DHPN value attribute with `TNC_TNCC_GetAttribute` but may not set this value with `TNC_TNCC_SetAttribute`. The IMC MUST provide a valid connection ID to `TNC_TNCC_GetAttribute`. The attribute value returned by the TNCC 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.

TNCCs are not required to implement the DHPN attribute, but they SHOULD do so if possible. IMCs MAY make use of this attribute when available but are not required to do so. Many TNCCs do not support this attribute; IMCs MUST work properly if a TNCC does not support it.

3.6.8.5 Has Long Types Attribute

The Has Long Types attribute allows a TNCC to indicate to the IMCs that a particular connection supports long message types. An IMC may get the value of the Has Long Types attribute with `TNC_TNCC_GetAttribute` but may not set this value with `TNC_TNCC_SetAttribute`. The

IMC MUST provide a valid connection ID to TNC_TNCC_GetAttribute. The attribute value returned by the TNCC will pertain to that connection. If the IMC does not provide a valid connection ID, the TNCC 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 IMC may call the TNC_TNCC_SendMessageLong function for this connection and should expect to receive calls for this connection to the IMC's TNC_IMC_ReceiveMessageLong function (if implemented by the IMC).

TNCCs MUST implement the Has Long Types attribute if they support the TNC_TNCC_SendMessageLong and TNC_IMC_ReceiveMessageLong functions. IMCs MAY make use of this attribute when available but are not required to do so. Many TNCCs do not support this attribute; IMCs MUST work properly if a TNCC does not support it.

3.6.8.6 Has Exclusive Attribute

The Has Exclusive attribute allows a TNCC to indicate to the IMCs that a particular connection supports exclusive delivery. An IMC may get the value of the Has Exclusive attribute with TNC_TNCC_GetAttribute but may not set this value with TNC_TNCC_SetAttribute. The IMC MUST provide a valid connection ID to TNC_TNCC_GetAttribute. The attribute value returned by the TNCC will pertain to that connection. If the IMC does not provide a valid connection ID, the TNCC 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 IMC may call the TNC_TNCC_SendMessageLong function for this connection and set the TNC_MESSAGE_FLAGS_EXCLUSIVE flag.

TNCCs MUST implement the Has Exclusive attribute if they support the TNC_TNCC_SendMessageLong and TNC_IMC_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 TNCCs may have a value of 0 for this attribute, depending on the IF-TNCCS protocol in use. IMCs MAY make use of this attribute when available but are not required to do so. Many TNCCs do not support this attribute; IMCs MUST work properly if a TNCC does not support it.

3.6.8.7 Has SOH Attribute

The Has SOH attribute allows a TNCC to indicate to the IMCs that a particular connection supports SOH functions like TNC_TNCC_SendMessageSOH. An IMC may get the value of the Has SOH attribute with TNC_TNCC_GetAttribute but may not set this value with TNC_TNCC_SetAttribute. The IMC MUST provide a valid connection ID to TNC_TNCC_GetAttribute. The attribute value returned by the TNCC will pertain to that connection. If the IMC does not provide a valid connection ID, the TNCC 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 TNCC MUST implement the SSOHR attribute and the TNC_TNCC_SendMessageSOH function. Further, the TNCC MUST be prepared to call an IMC's TNC_IMC_ReceiveMessageSOH function to deliver messages if the IMC implements this function. An IMC may therefore call the TNC_TNCC_SendMessageSOH function for this connection and should expect to receive calls for this connection to the IMC's `TNC_IMC_ReceiveMessageSOH` function (if implemented by the IMC). If an IMC does not implement `TNC_IMC_ReceiveMessageSOH` and an IF-TNCCS-SOH connection is started, messages will be delivered to that IMC using `TNC_IMC_ReceiveMessage` instead, as described in section 3.8.5.

A TNCC MUST implement the Has SOH attribute if it supports the `TNC_TNCC_SendMessageSOH` and `TNC_IMC_ReceiveMessageSOH` functions. IMCs MAY use the Has SOH attribute when available but are not required to do so. Many TNCCs do not support this attribute; IMCs MUST work properly if a TNCC does not support it.

3.6.8.8 SOHR Attribute

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

The attribute value for the SOHR attribute is a sequence of bytes that contains the binary value of the SoHR sent by the TNCS on this connection, as described in section 3.6.1.1 of [9]. No padding or encapsulation should be added, so the first byte in the value is the first byte in the SoHR (the first byte in the SoHR's header) and the last byte is the last byte in the SoHR (the last byte in the last SoHRReportEntry). The length for this attribute will vary depending on its contents.

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

3.6.8.9 SSOHR Attribute

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

The attribute value for the SSOHR attribute is a sequence of bytes that contains the binary value of the SSoHR field in the SoHR sent by the TNCS on this connection, as described in section 3.6.1.2 of [9]. No padding or encapsulation should be added, so the first byte in the value is the first byte in the SSoHR (the first byte in the SSoHR's System-Health-Id attribute) and the last byte is the last byte in the SSoHR (the last byte in the Vendor-Specific attribute that contains the SSoHR attributes). The length for this attribute will vary depending on its contents.

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

3.6.8.10 IF-TNCCS Protocol Attribute

The IF-TNCCS Protocol attribute allows IMCs to determine which IF-TNCCS Protocol or similar protocol is being used for a particular connection. An IMC may get the value of the IF-TNCCS Protocol attribute with `TNC_TNCC_GetAttribute` but may not set this value with `TNC_TNCC_SetAttribute`. The IMC MUST provide a valid connection ID to `TNC_TNCC_GetAttribute`. The attribute value returned by the TNCC will pertain to that connection. If the IMC does not provide a valid connection ID, the TNCC 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 [11], IF-TNCCS 1.1 [12], or IF-TNCCS 2.0 [10] and "IF-TNCCS-SOH" for connections that use IF-TNCCS-SOH 1.0 [9]. 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, TNCCs 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 should start with the vendor's SMI Private Enterprise Number expressed in ASCII with no preceding 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 IMCs should be prepared to receive non-ASCII characters. Despite the flexibility described in this paragraph, TNCC vendors are encouraged to restrict themselves to the standardized IF-TNCCS protocols as much as possible.

If the TNCC 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 TNCC SHOULD return `TNC_RESULT_INVALID_PARAMETER`. The IMC may wish to check back later in the lifecycle of the connection, since this information might be available later. In spite of this provision, TNCCs SHOULD have this information available for all connections before calling any IMC functions with that connection ID. TNCCs 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` and then the connection ID is used again and the connection state changes to `TNC_CONNECTION_STATE_CREATE`).

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

3.6.8.11 IF-TNCCS Version Attribute

The IF-TNCCS Version attribute allows IMCs to determine the version of the IF-TNCCS Protocol or similar protocol that is being used for a particular connection. An IMC may get the value of the IF-TNCCS Version attribute with `TNC_TNCC_GetAttribute` but may not set this value with `TNC_TNCC_SetAttribute`. The IMC MUST provide a valid connection ID to `TNC_TNCC_GetAttribute`. The attribute value returned by the TNCC will pertain to that connection. If the IMC does not provide a valid connection ID, the TNCC 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. Here is a list of strings to be used for existing protocols: "1.0" for connections that use IF-TNCCS 1.0 [11] or IF-TNCCS-SOH 1.0 [9], "1.1" for connections that use IF-TNCCS 1.1 [12], and "2.0" for connections that use IF-TNCCS 2.0 [10]. 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, TNCCs 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 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 IMCs MUST 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 TNCC 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 TNCC SHOULD return `TNC_RESULT_INVALID_PARAMETER`. The IMC may wish to check back later in the lifecycle of the connection, since this information might be available later. In spite of this provision, TNCC's SHOULD have this information available for all connections before calling any IMC functions with that connection ID. TNCCs 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` and then the connection ID is used again and the connection state changes to `TNC_CONNECTION_STATE_CREATE`).

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

3.6.8.12 IF-T Protocol Attribute

The IF-T Protocol attribute allows IMCs to determine which IF-T Protocol or other transport protocol is being used for a particular connection. An IMC may get the value of the IF-T Protocol attribute with `TNC_TNCC_GetAttribute` but may not set this value with `TNC_TNCC_SetAttribute`. The IMC MUST provide a valid connection ID to `TNC_TNCC_GetAttribute`. The attribute value returned by the TNCC will pertain to that connection. If the IMC does not provide a valid connection ID, the TNCC 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: "IF-T for Tunneled EAP" for connections that use IF-T for Tunneled EAP Methods 1.0 [7] or IF-T for Tunneled EAP Methods 1.1 [8], “IF-T for TLS” for connections that use IF-T Binding to TLS [15] [19]. In addition, connections that use PEAP [13] 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, TNCCs 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 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-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 with no preceding 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 IMCs should be prepared to receive non-ASCII characters. Despite the flexibility described in this paragraph, TNCC vendors are encouraged to restrict themselves to the standardized IF-T protocols as much as possible.

If the TNCC 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 TNCC SHOULD return `TNC_RESULT_INVALID_PARAMETER`. The IMC may wish to check back later in the lifecycle of the connection, since this information might be available later. In spite of this provision, TNCCs SHOULD have this information available for all connections before calling any IMC functions with that connection ID. TNCCs 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` and then the connection ID is used again and the connection state changes to `TNC_CONNECTION_STATE_CREATE`).

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

3.6.8.13 IF-T Version Attribute

The IF-T Version attribute allows IMCs to determine the version of the IF-T Protocol or other transport protocol that is being used for a particular connection. An IMC may get the value of the IF-T Version attribute with `TNC_TNCC_GetAttribute` but may not set this value with `TNC_TNCC_SetAttribute`. The IMC MUST provide a valid connection ID to `TNC_TNCC_GetAttribute`. The attribute value returned by the TNCC will pertain to that connection. If the IMC does not provide a valid connection ID, the TNCC 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. Here is a list of strings to be used for existing protocols: "1.0" for connections that use IF-T for Tunneled EAP Methods 1.0 [7] and "1.1" for connections that use IF-T for Tunneled EAP Methods 1.1 [8]. 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, TNCCs 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 IMCs 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 TNCC 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 TNCC SHOULD return `TNC_RESULT_INVALID_PARAMETER`. The IMC may wish to check back later in the lifecycle of the connection, since this information might be available later. In spite of this provision, TNCC's SHOULD have this information available for all connections before calling any IMC functions with that connection ID. TNCCs 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` and then the connection ID is used again and the connection state changes to `TNC_CONNECTION_STATE_CREATE`).

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

3.6.8.14 IMC Supports TNCS First Attribute

The IMC Supports TNCS First attribute allows an IMC to indicate to the TNCC that the IMC supports having the TNCS send the first batch in an exchange. An IMC may set the value of the IMC Supports TNCS First attribute with `TNC_TNCC_SetAttribute` and may get this value with `TNC_TNCC_GetAttribute`. The IMC MUST provide `TNC_CONNECTIONID_ANY` as the connection ID when getting or setting this attribute value, since the IMC's ability to support the TNCS sending the first batch is not particular to a single connection. If the IMC does not provide `TNC_CONNECTIONID_ANY` as the connection ID, the TNCC SHOULD return `TNC_RESULT_INVALID_PARAMETER`. An IMC that is going to set the IMC Supports TNCS First attribute MUST set this attribute as soon as possible; once this attribute has been set, the IMC MUST NOT change its value. These prohibitions should ensure that the IMC receives the proper messages in each handshake.

The attribute value for IMC Supports TNCS First is one byte with a value of zero (0) if the IMC does not support having the TNCS send the first batch in an exchange and a value of one (1) if the IMC does support that. The length for this attribute MUST always be 1. If an IMC does not set this attribute, its value is a single byte with value 0.

A TNCC that supports IF-TNCCS 2.0 MUST support this attribute and MUST allow each IMC to set a different value for the attribute, since some IMCs can support having the TNCS send the first message in an exchange and some cannot. A TNCC that does not support this attribute SHOULD return the result code `TNC_RESULT_INVALID_PARAMETER` when an IMC attempts to set or get this attribute, since the TNCC will not recognize the attribute ID.

If the value of this attribute is 0, the TNCC may still engage in an exchange where the TNCS sends the first message. However, the TNCC MUST NOT deliver any IMC-IMV messages received in that batch to this particular IMC. For that IMC, it should appear that the TNCC is sending the first message.

IMCs MAY use this attribute when available but are not required to do so. Many TNCCs do not support this attribute; IMCs MUST work properly if a TNCC does not support it.

3.6.8.15 Primary IMC ID Attribute

The Primary IMC ID attribute indicates the unique identifier of the IMC assigned by the TNCC when the TNCC loads this IMC. An IMC may get the value of the Primary IMC ID attribute with `TNC_TNCC_GetAttribute` but may not set this value with `TNC_TNCC_SetAttribute`.

The attribute value for this attribute is a `TNC_IMCID`.

TNCCs MUST implement the Primary IMC ID attribute if they support the `TNC_TNCC_SendMessageLong` and `TNC_IMC_ReceiveMessageLong` functions, because the use of this attribute is essential to the proper use of those functions. IMCs MAY make use of this attribute when available but are not required to do so. Many TNCCs do not support this attribute; IMCs MUST work properly if a TNCC does not support it. IMCs that use the DLL binding will generally not need this attribute, since the TNCC passes the primary IMC ID to the IMC when it calls the `TNC_IMC_Initialize` function. The main purpose of this function is for IMCs that use the Java binding and support multiple IMC IDs, since the TNCC does not generally pass the primary IMC ID to the IMC with the Java binding.

3.6.8.16 TLS-Unique Attribute

The TLS-Unique attribute allows a TNCC that supports TLS-Unique (as described in IF-T for Tunneled EAP Methods and IF-T for TLS) to provide to a PTS-IMC or other IMCs a TLS-Unique value obtained from the underlying TLS session, so that this value can be used in the TPM_Quote operation. An IMC may get the value of the TLS-Unique value with `TNC_TNCC_GetAttribute` but may not set this value with `TNC_TNCC_SetAttribute`. The IMC MUST provide a valid connection ID to `TNC_TNCC_GetAttribute`. The attribute value returned by the TNCC will pertain to that connection. If the IMC does not provide a valid connection ID or provides a connection ID for a connection that does not have a TLS-Unique value, the TNCC SHOULD return `TNC_RESULT_INVALID_PARAMETER`, since different connections will have different TLS-Unique values (and some may have none).

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.

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

3.7 Mandatory and Optional Functions

Some of the functions in the IF-IMC API are marked as mandatory below. Mandatory functions MUST be implemented. The rest of the functions in the IF-IMC API are marked as optional and need not be implemented. An IMC or TNC Client 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 IMC Functions

These functions are implemented by the IMC and called by the TNC Client.

3.8.1 TNC_IMC_Initialize (MANDATORY)

```
TNC_Result TNC_IMC_Initialize(
      /*in*/  TNC_IMCID imcID,
      /*in*/  TNC_Version minVersion,
      /*in*/  TNC_Version maxVersion,
      /*out*/ TNC_Version *pOutActualVersion);
```

Description:

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

The TNC Client MUST NOT call any other IF-IMC API functions for an IMC until it has successfully completed a call to `TNC_IMC_Initialize`(). Once a call to this function has completed successfully, this function MUST NOT be called again for a particular IMC-TNCC pair until a call to `TNC_IMC_Terminate` has completed successfully.

The TNC Client MUST set `minVersion` to the minimum IF-IMC API version number that it supports and MUST set `maxVersion` to the maximum API version number that it supports. The TNC Client also MUST set `pOutActualVersion` so that the IMC 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 IMC MUST check these to determine whether there is an API version number that it supports in this range. If not, the IMC MUST return `TNC_RESULT_NO_COMMON_VERSION`. Otherwise, the IMC SHOULD select a mutually supported version number, store that version number at `pOutActualVersion`, and initialize the IMC. If the initialization completes successfully, the IMC SHOULD return `TNC_RESULT_SUCCESS`. Otherwise, it SHOULD return another result code.

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

Input Parameter	**Description**
`imcID`	IMC ID assigned by TNCC
`minVersion`	Minimum API version supported by TNCC
`maxVersion`	Maximum API version supported by TNCC

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 IMC and TNC Client
`TNC_RESULT_ALREADY_INITIALIZED`	`TNC_IMC_Initialize` has already been called and `TNC_IMC_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_IMC_NotifyConnectionChange (OPTIONAL)

```
TNC_Result TNC_IMC_NotifyConnectionChange(
      /*in*/  TNC_IMCID imcID,
      /*in*/  TNC_ConnectionID connectionID
      /*in*/  TNC_ConnectionState newState);
```

Description:

The TNC Client calls this function to inform the IMC that the state of the network connection identified by `connectionID` has changed to `newState`. Section 3.6.4 lists all the possible values of `newState` for this version of the IF-IMC API. The TNCC MUST NOT use any other values with this version of IF-IMC.

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

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

If the state is `TNC_CONNECTION_STATE_ACCESS_ALLOWED` or `TNC_CONNECTION_STATE_ACCESS_ISOLATED`, the IMC SHOULD proceed with any remediation instructions received during the Integrity Check Handshake. However, the IMC SHOULD be prepared for delays in network access or even complete denial of network access, even in these cases. Network access will often be delayed for a few seconds while an IP address is acquired. And network access may be denied if the NAA overrides the TNCS Action Recommendation reflected in the newState value.

If the state is `TNC_CONNECTION_STATE_ACCESS_NONE`, the IMC MAY discard any remediation instructions received during the Integrity Check Handshake or it MAY follow them if possible.

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 IMC SHOULD discard any state pertaining to this network connection and MUST NOT pass this network connection ID to the TNC Client after this function returns (unless the TNCC later creates another network connection with the same network connection ID).

In the `imcID` parameter, the TNCC MUST pass the IMC ID value provided to `TNC_IMC_Initialize`. In the `connectionID` parameter, the TNCC MUST pass a valid network connection ID. IMCs MAY check these values to make sure they are valid and return an error if not, but IMCs are not required to make these checks. In the `newState` parameter, the TNCC MUST pass one of the values listed in section 3.6.4.

Input Parameter	**Description**
`imcID`	IMC ID assigned by TNCC
`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_IMC_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_IMC_BeginHandshake (MANDATORY)

```
TNC_Result TNC_IMC_BeginHandshake(
      /*in*/  TNC_IMCID imcID,
      /*in*/  TNC_ConnectionID connectionID);
```

Description:

The TNC Client calls this function to indicate that an Integrity Check Handshake is beginning and solicit messages from IMCs for the first batch. The IMC 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.

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

All IMCs MUST implement this function.

In the `imcID` parameter, the TNCC MUST pass the IMC ID value provided to `TNC_IMC_Initialize`. In the `connectionID` parameter, the TNCC MUST pass a valid network connection ID. IMCs MAY check these values to make sure they are valid and return an error if not, but IMCs are not required to make these checks.

Input Parameter	**Description**
`imcID`	IMC ID assigned by TNCC
`connectionID`	Network connection ID on which message was received

Result Code	**Condition**
`TNC_RESULT_SUCCESS`	Success
`TNC_RESULT_NOT_INITIALIZED`	`TNC_IMC_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_IMC_ReceiveMessage (OPTIONAL)

```
TNC_Result TNC_IMC_ReceiveMessage(
      /*in*/  TNC_IMCID imcID,
      /*in*/  TNC_ConnectionID connectionID,
      /*in*/  TNC_BufferReference message,
      /*in*/  TNC_UInt32 messageLength,
      /*in*/  TNC_MessageType messageType);
```

Description:

The TNC Client calls this function to deliver a message to the IMC. 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 IMV (or a TNCS or other party acting as an IMV).

If IF-TNCCS-SOH is used for a connection and the IMC and TNCC implement `TNC_IMC_ReceiveMessageSOH`, the TNCC SHOULD use that function to deliver the contents of SOHRReportEntries instead of using `TNC_IMC_ReceiveMessage`. However, if IF-TNCCS-

SOH is used for a connection but either the IMC or the TNCC does not implement `TNC_IMC_ReceiveMessageSOH`, `TNC_IMC_ReceiveMessage` SHOULD be used instead. For each SOHResponseEntry, the value contained in the first System-Health-ID attribute should be compared to the `TNC_MessageType` values previously supplied in the IMC's most recent call to `TNC_TNCC_ReportMessageTypes` or `TNC_TNCC_ReportMessageTypesLong`. The SOHResponseEntry should be delivered to each IMC that has a match. If an IMC that does not support `TNC_IMC_ReceiveMessageSOH` (or when the TNCC does not support that function), `TNC_IMC_ReceiveMessage` SHOULD be employed. In that case, the TNCC 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 SOHResponseEntry MUST NOT be delivered to this IMC. The TNCC MUST pass in the `messageLength` parameter the number of octets in this buffer. And the TNCC MUST pass in the `messageType` parameter the value contained in the first System-Health-ID attribute.

The IMC 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 IMC functions, the IMC SHOULD NOT wait a long time (definitely not more than a second) before returning from `TNC_IMC_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 IMC should implement this function if it wants to receive messages. Simple IMCs that only send messages need not implement this function. The IMC MUST NOT ever modify the buffer contents and MUST NOT access the buffer after `TNC_IMC_ReceiveMessage` has returned. If the IMC wants to retain the message, it should copy it before returning from `TNC_IMC_ReceiveMessage`.

In the `imcID` parameter, the TNCC MUST pass the IMC ID value provided to `TNC_IMC_Initialize`. In the `connectionID` parameter, the TNCC MUST pass a valid network connection ID. In the `message` parameter, the TNCC MUST pass a reference to a buffer containing the message being delivered to the IMC. In the `messageLength` parameter, the TNCC 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 TNCC MUST pass the type of the message. This value MUST match one of the `TNC_MessageType` values previously supplied by the IMC to the TNCC in the IMC's most recent call to `TNC_TNCC_ReportMessageTypes` or `TNC_TNCC_ReportMessageTypesLong`. IMCs MAY check these parameters to make sure they are valid and return an error if not, but IMCs are not required to make these checks.

Input Parameter	**Description**
`imcID`	IMC ID assigned by TNCC
`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_IMC_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_IMC_ReceiveMessageSOH (OPTIONAL)

```
TNC_Result TNC_IMC_ReceiveMessageSOH(
      /*in*/  TNC_IMCID imcID,
      /*in*/  TNC_ConnectionID connectionID,
      /*in*/  TNC_BufferReference sohrReportEntry,
      /*in*/  TNC_UInt32 sohrRELength,
      /*in*/  TNC_MessageType systemHealthID);
```

Description:

The TNC Client calls this function (if supported by the TNCC and implemented by an IMC) to deliver an SOHRReportEntry message to the IMC. This allows an IMC to receive the entire SOHRReportEntry instead of just the contents of the Vendor-Specific attribute, as would be the case if the `TNC_IMC_ReceiveMessage` function was used.

A TNCC that supports IF-TNCCS-SOH SHOULD support this function; a TNCC that supports this function SHOULD call this function for a particular IMC and connection instead of calling `TNC_IMC_ReceiveMessage` if the IMC implements this function and the connection uses IF-TNCCS-SOH. A TNCC MUST NOT call this function for a particular IMC and connection if the IMC does not implement this function or the connection does not use IF-TNCCS-SOH. A TNCC will often find that some IMCs loaded by that TNCC implement this function and some do not. In that case, when the TNCC is handling a connection that uses IF-TNCCS-SOH, the TNCC would call `TNC_IMC_ReceiveMessageSOH` for the IMCs that implement this function and `TNC_IMC_ReceiveMessage` for the IMCs that do not implement `TNC_IMC_ReceiveMessageSOH`. A TNCC MUST NOT call both `TNC_IMC_ReceiveMessage` and `TNC_IMC_ReceiveMessageSOH` for a single SOHRReportEntry for a single IMC.

IMCs are not required to implement this function. An IMC should implement `TNC_IMC_ReceiveMessageSOH` if it wants to receive the entire SOHRReportEntry instead of just the Vendor-Specific attribute when IF-TNCCS-SOH is used. However, IMCs should recognize that many TNCCs do not support IF-TNCCS-SOH, and some TNCCs that do support IF-TNCCS-SOH will not support this function. Therefore, IMCs should be prepared to receive messages via `TNC_IMC_ReceiveMessage` in some cases when IF-TNCC-SOH is used. Simple IMCs that only send messages need not implement this function. IMC implementers may find that implementing `TNC_IMC_ReceiveMessageSOH` is more complex than implementing `TNC_IMC_ReceiveMessage`, since `TNC_IMC_ReceiveMessageSOH` must parse the SOHRReportEntry, while `TNC_IMC_ReceiveMessage` only needs to parse the contents of the Vendor-Specific attribute within the SOHRReportEntry. For many IMCs, the contents of the Vendor-Specific attribute is all that they need. Therefore, IMCs are not required to implement `TNC_IMC_ReceiveMessageSOH`.

The content of the SOHRReportEntry is contained in the buffer referenced by `sohrReportEntry` and contains the number of octets (bytes) indicated by `sohrRELength`. The content of the first System-Health-ID attribute in the SOHRReportEntry is indicated by `systemHealthID`.

The IMC MUST NOT send any IMC-IMV messages on the connection after this function is called, since IF-TNCCS-SOH supports only one round-trip.

As with all IMC functions, the IMC SHOULD NOT wait a long time (definitely not more than a second) before returning from `TNC_IMC_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 IMC MUST NOT ever modify the buffer contents and MUST NOT access the buffer after `TNC_IMC_ReceiveMessageSOH` has returned. If the IMC wants to retain the contents of the SOHRReportEntry, it should copy it before returning from `TNC_IMC_ReceiveMessageSOH`.

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

Input Parameter	**Description**
`imcID`	IMC ID assigned by TNCC
`connectionID`	Network connection ID on which message was received
`sohrReportEntry`	Reference to buffer containing SOHRReportEntry
`sohrRELength`	Number of octets in SOHRReportEntry
`systemHealthID`	Content of first System-Health-ID attribute in SOHRReportEntry

Result Code	**Condition**
`TNC_RESULT_SUCCESS`	Success
`TNC_RESULT_NOT_INITIALIZED`	`TNC_IMC_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_IMC_ReceiveMessageLong (OPTIONAL)

```
TNC_Result TNC_IMC_ReceiveMessageLong(
      /*in*/  TNC_IMCID imcID,
      /*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 sourceIMVID,
      /*in*/  TNC_UInt32 destinationIMCID);
```

Description:

The TNC Client calls this function to deliver a message to the IMC. This function provides several features that `TNC_IMC_ReceiveMessage` does not: longer (32 bit) vendor ID and message subtype fields, a `messageFlags` field, and the ability to specify a source IMV ID and destination IMC 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 IMV (or a TNCS or other party acting as an IMV). Any flags associated with the message are included in the `messageFlags` parameter. The `sourceIMVID` and `destinationIMCID` parameters indicate the IMV ID of the IMV that sent this message (if available) and either the IMC ID of the intended recipient (if the EXCL flag is set), or the IMC ID in response to whose message or messages this message was sent. If the EXCL flag is set, `destinationIMCID` MUST be either the primary IMC ID provided to the IMC in `TNC_IMC_Initialize`, or an additional IMC ID reserved when the IMC requested the TNCC to do so by calling `TNC_TNCC_ReserveAdditionalIMCID`. If the EXCL flag is not set, then `destinationIMCID` MAY be set to the wild card `TNC_IMCID_ANY`.

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

The IMC 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 IMC functions, the IMC SHOULD NOT wait a long time (definitely not more than a second) before returning from `TNC_IMC_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 IMCs are not required to implement it. An IMC should implement this function if it wants to receive messages with long types, or flags, or source IMV IDs and destination IMC IDs. Simple IMCs that only send messages need not implement this function. Since this function was not included in IF-IMC 1.0 through 1.2, many IMCs do not implement it. TNCCs MUST work properly if an IMC does not implement this function. The TNCC SHOULD use dynamic function binding (on platforms where that is available) to determine whether the IMC implements this function.

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

In the `imcID` parameter, the TNCC MUST pass the IMC ID value provided to `TNC_IMC_Initialize`. In the `connectionID` parameter, the TNCC MUST pass a valid network connection ID. In the `message` parameter, the TNCC MUST pass a reference to a buffer containing the message being delivered to the IMC. In the `messageLength` parameter, the TNCC 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 TNCC MUST pass the vendor ID and message subtype of the message. These values MUST match one of the values previously supplied by the IMC to the TNCC in the IMC's most recent call to `TNC_TNCC_ReportMessageTypes` or `TNC_TNCC_ReportMessageTypesLong`. The TNCC 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. IMCs MAY check these parameters to make sure they are valid and return an error if not, but IMCs 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 TNCC MUST process that flag itself to ensure that a message with this flag set is only delivered to the intended recipient.

Input Parameter	**Description**
`imcID`	IMC ID assigned by TNCC
`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
`sourceIMVID`	Source IMV ID for 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_OTHER`	Unspecified non-fatal error

TNC_RESULT_FATAL	Unspecified fatal error
Other result codes	Other non-fatal error

3.8.7 TNC_IMC_BatchEnding (OPTIONAL)

```
TNC_Result TNC_IMC_BatchEnding(
      /*in*/  TNC_IMCID imcID,
      /*in*/  TNC_ConnectionID connectionID);
```

Description:

The TNC Client calls this function to notify IMCs that all IMV messages received in a batch have been delivered and this is the IMC's last chance to send a message in the batch of IMC messages currently being collected. An IMC MAY implement this function if it wants to perform some actions after all the IMV messages received during a batch have been delivered (using `TNC_IMC_ReceiveMessage`, `TNC_IMC_ReceiveMessageSOH`, or `TNC_IMC_ReceiveMessageLong`). This is especially useful for IMCs that have included a wildcard in the list of message types reported using `TNC_TNCC_ReportMessageTypes` or `TNC_TNCC_ReportMessageTypesLong`.

An IMC MAY call `TNC_TNCC_SendMessage`, `TNC_TNCC_SendMessageSOH`, or `TNC_TNCC_SendMessageLong` from this function. As with all IMC functions, the IMC SHOULD NOT wait a long time (definitely not more than a second) before returning from `TNC_IMC_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 `imcID` parameter, the TNCC MUST pass the IMC ID value provided to `TNC_IMC_Initialize`. In the `connectionID` parameter, the TNCC MUST pass a valid network connection ID. IMCs MAY check these values to make sure they are valid and return an error if not, but IMCs are not required to make these checks.

Input Parameter	**Description**
imcID	IMC ID assigned by TNCC
connectionID	Network connection ID for which a batch is ending

Result Code	**Condition**
TNC_RESULT_SUCCESS	Success
TNC_RESULT_NOT_INITIALIZED	`TNC_IMC_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_IMC_Terminate (OPTIONAL)

```
TNC_Result TNC_IMC_Terminate(
      /*in*/  TNC_IMCID imcID);
```

Description:

The TNC Client calls this function to close down the IMC when all work is complete or the IMC reports `TNC_RESULT_FATAL`. Once a call to `TNC_IMC_Terminate` is made, the TNC Client MUST NOT call the IMC except to call `TNC_IMC_Initialize` (which may not succeed if the IMC cannot reinitialize itself). Even if the IMC returns an error from this function, the TNC Client MAY continue with its unload or shutdown procedure.

In the `imcID` parameter, the TNCC MUST pass the IMC ID value provided to `TNC_IMC_Initialize`. IMCs MAY check if `imcID` matches the value previously passed to `TNC_IMC_Initialize` and return `TNC_RESULT_INVALID_PARAMETER` if not, but they are not required to make this check.

Input Parameter	**Description**
`imcID`	IMC ID assigned by TNCC

Result Code	**Condition**
`TNC_RESULT_SUCCESS`	Success
`TNC_RESULT_NOT_INITIALIZED`	`TNC_IMC_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 Client Functions

These functions are implemented by the TNC Client and called by the IMC.

3.9.1 TNC_TNCC_ReportMessageTypes (MANDATORY)

```
TNC_Result TNC_TNCC_ReportMessageTypes(
      /*in*/  TNC_IMCID imcID,
      /*in*/  TNC_MessageTypeList supportedTypes,
      /*in*/  TNC_UInt32 typeCount);
```

Description:

An IMC calls this function to inform a TNCC about the set of message types the IMC is able to receive. Often, the IMC will call this function from `TNC_IMC_Initialize`. With the Windows DLL binding or UNIX/Linux Dynamic Linkage binding, `TNC_TNCC_ReportMessageTypes` will typically be called from `TNC_IMC_ProvideBindFunction` since an IMC cannot call the TNCC with those platform bindings until `TNC_IMC_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 imcID, the IMC MUST pass the value provided to `TNC_IMC_Initialize`. TNCCs MAY check if `imcID` matches the value previously passed to `TNC_IMC_Initialize` and return `TNC_RESULT_INVALID_PARAMETER` if not, but they are not required to make this check.

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

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

If an IMC 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 IMC. This message type is `0xffffffff`. 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, except for messages marked for exclusive delivery to another IMC. Each time a particular IMC calls `TNC_TNCC_ReportMessageTypes` or `TNC_TNCC_ReportMessageTypesLong`, the message type list supplied in the latest call supplants the message type lists supplied by that IMC in earlier calls.

Input Parameter	**Description**
`imcID`	IMC ID assigned by TNCC
`supportedTypes`	Reference to list of message types supported by IMC
`typeCount`	Number of message types supported by IMC

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_TNCC_ReportMessageTypesLong (OPTIONAL)

```
TNC_Result TNC_TNCC_ReportMessageTypesLong(
      /*in*/  TNC_IMCID imcID,
      /*in*/  TNC_VendorIDList supportedVendorIDs,
      /*in*/  TNC_MessageSubtypeList supportedSubtypes,
      /*in*/  TNC_UInt32 typeCount);
```

Description:

An IMC calls this function to inform a TNCC about the set of message types the IMC is able to receive. This function supports long message types, unlike `TNC_TNCC_ReportMessageTypes`.

Often, the IMC will call this function from `TNC_IMC_Initialize`. With the Windows DLL binding or UNIX/Linux Dynamic Linkage binding, `TNC_TNCC_ReportMessageTypesLong` will often be called from `TNC_IMC_ProvideBindFunction`, since an IMC cannot call the TNCC with those platform bindings until `TNC_IMC_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 IMC is able to receive.

In the imcID parameter, the IMC MUST pass the value provided to `TNC_IMC_Initialize`. TNCCs MAY check if `imcID` matches the value previously passed to `TNC_IMC_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 TNCCs. However, a TNCC 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. IMCs should recognize that many TNCCs do not support long types and therefore will not support this function. In those cases, IMCs should be prepared to use the `TNC_TNCC_ReportMessageTypes` function. Simple IMCs that do not receive messages need not call this function.

The TNC Client MUST NOT ever modify the list of message types and MUST NOT access this list after `TNC_TNCC_ReportMessageTypesLong` has returned. Generally, the TNC Client will copy the contents of this list before returning from this function. TNC Clients MUST support any message type.

If an IMC 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 IMC. 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, except for messages marked for exclusive delivery to another IMC. Each time a particular IMC calls `TNC_TNCC_ReportMessageTypes` or `TNC_TNCC_ReportMessageTypesLong`, the message type list supplied in the latest call supplants the message type lists supplied by that IMC in earlier calls.

Input Parameter	**Description**
`imcID`	IMC ID assigned by TNCC
`supportedVendorIDs`	Reference to list of Vendor IDs supported by IMC
`supportedSubtypes`	Reference to list of Message Subtypes supported by IMC
`typeCount`	Number of message types supported by IMC

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_TNCC_SendMessage (MANDATORY)

```
TNC_Result TNC_TNCC_SendMessage(
      /*in*/  TNC_IMCID imcID,
      /*in*/  TNC_ConnectionID connectionID,
      /*in*/  TNC_BufferReference message,
      /*in*/  TNC_UInt32 messageLength,
      /*in*/  TNC_MessageType messageType);
```

Description:

An IMC calls this function to give a message to the TNCC 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 `imcID` parameter, the IMC MUST pass the value provided to `TNC_IMC_Initialize`. In the `connectionID` parameter, the IMC MUST pass a valid network connection ID. TNCCs MAY check these values to make sure they are valid and return an error if not, but TNCCs are not required to make these checks.

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

The IMC MUST NOT call this function unless it has received a call to `TNC_IMC_BeginHandshake`, `TNC_IMC_ReceiveMessage`, `TNC_IMC_ReceiveMessageSOH`, `TNC_IMC_ReceiveMessageLong`, or `TNC_IMC_BatchEnding` for this connection and the IMC has not yet returned from that function. If the IMC violates this prohibition, the TNCC SHOULD return `TNC_RESULT_ILLEGAL_OPERATION`. If an IMC really wants to communicate with an IMV at another time, it should call `TNC_TNCC_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 TNCC will return `TNC_RESULT_ILLEGAL_OPERATION` from `TNC_TNCC_SendMessage`. If the TNCC supports limiting the message size or number of round trips, the TNCC MUST return TNC_RESULT_EXCEEDED_MAX_MESSAGE_SIZE or TNC_RESULT_EXCEEDED_MAX_ROUND_TRIPS respectively if the limits are exceeded. An IMC can get the Maximum Message Size and Maximum Number of Round Trips by using the related Attribute ID within the TNC_TNCC_GetAttribute function. The IMC SHOULD adapt its behavior to accommodate these limitations if available.

The TNC Client MUST support any message type. However, the IMC 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 IMC violates this prohibition, the TNCC SHOULD return `TNC_RESULT_INVALID_PARAMETER`.

Input Parameter	**Description**
`imcID`	IMC ID assigned by TNCC
`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_OTHER`	Unspecified non-fatal error
`TNC_RESULT_FATAL`	Unspecified fatal error
Other result codes	Other non-fatal error

3.9.4 TNC_TNCC_RequestHandshakeRetry (MANDATORY)

```
TNC_Result TNC_TNCC_RequestHandshakeRetry(
      /*in*/  TNC_IMCID imcID,
      /*in*/  TNC_ConnectionID connectionID,
      /*in*/  TNC_RetryReason reason);
```

Description:

An IMC calls this function to ask a TNCC to retry an Integrity Check Handshake. The IMC MUST pass its IMC ID as the `imcID` parameter, a network connection ID as the `connectionID` parameter, and one of the handshake retry reasons listed in section 3.6.5 as the `reason` parameter. If the network connection ID is `TNC_CONNECTIONID_ANY`, then the IMC requests an Integrity Check Handshake retry on all current network connections.

TNCCs MAY check the parameters to make sure they are valid and return an error if not, but TNCCs are not required to make these checks. The `reason` parameter explains why the IMC is requesting a handshake retry. The TNCC MAY use this in deciding whether to attempt the handshake retry. As noted in section 2.10.3, TNCCs are not required to honor IMC requests for handshake retry (especially since handshake retry may not be possible or may interrupt network connectivity). An IMC MAY call this function at any time, even if an Integrity Check Handshake is currently underway. This is useful if the IMC suddenly gets important information but has already finished its dialog with the IMV, for instance. As always, the TNCC is not required to honor the request for handshake retry.

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

Input Parameter	Description
`imcID`	IMC ID assigned by TNCC
`connectionID`	Network connection ID for which handshake retry is requested
`reason`	Reason why handshake retry is requested

Result Code	Condition
TNC_RESULT_SUCCESS	TNCC intends to retry the handshake
TNC_RESULT_CANT_RETRY	TNCC cannot attempt the handshake retry
TNC_RESULT_WONT_RETRY	TNCC 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

3.9.5 TNC_TNCC_GetAttribute (OPTIONAL)

```
TNC_Result TNC_TNCC_GetAttribute(
      /*in*/  TNC_IMCID imcID,
      /*in*/  TNC_ConnectionID connectionID,
      /*in*/  TNC_AttributeID attributeID,
      /*in*/  TNC_UInt32 bufferLength,
      /*out*/ TNC_BufferReference buffer,
      /*out*/ TNC_UInt32 *pOutValueLength);
```

Description:

An IMC calls this function to get the value of an attribute associated with a connection or with the TNCC as a whole. This function is optional. The TNCC is not required to implement it. Since this function was not included in IF-IMC 1.0, many TNCCs do not support it. IMCs MUST work properly if a TNCC does not implement this function.

The IMC MUST pass its IMC ID as the `imcID` parameter, a standard or vendor-specific attribute ID as the `attributeID` parameter, and a valid network connection ID as the `connectionID` parameter. If the IMC passes a valid connection ID, the TNCC SHOULD provide the attribute value for the specified connection (if any). If the IMC passes `TNC_CONNECTIONID_ANY`, the TNCC SHOULD provide the value for the TNCC (if any). If the TNCC does not recognize the attribute ID or connection ID, it SHOULD return the `TNC_RESULT_INVALID_PARAMETER` result code. If the TNCC 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 IMC MUST set `pOutValueLength` so that the TNCC 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 TNCC 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 IMC passes 0 for the `bufferLength` parameter, the TNCC MUST ignore the value of the `buffer` parameter. If the IMC passes a non-zero value for the `bufferLength` parameter, the IMC MUST set the `buffer` parameter so that the TNCC can use it as an output parameter to provide the requested attribute value. The IMC MUST provide enough storage for at least `bufferLength` bytes to be stored via the `buffer` parameter.

The TNCC 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 TNCC MUST NOT store any data via the `buffer` parameter. Instead, it MUST simply store the length via the `pOutValueLength` parameter. This allows the IMC to recognize that more storage space is needed. In either case, a result code of `TNC_RESULT_SUCCESS` SHOULD be returned.

The TNCC 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.

Some attributes have different values for different IMCs (e.g. the IMC Supports TNCS First attribute). These attributes are identified as such when they are defined. If a TNCC supports one of these attributes, the TNCC MUST maintain separate values for the attribute per IMC (and per connection if the attribute is a connection-specific attribute).

Input Parameter	**Description**
`imcID`	IMC ID assigned by TNCC
`connectionID`	Network connection ID for which an attribute value is desired (or `TNC_CONNECTIONID_ANY` to get information for the TNCC 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_IMC_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

3.9.6 TNC_TNCC_SetAttribute (OPTIONAL)

```
TNC_Result TNC_TNCC_SetAttribute(
      /*in*/  TNC_IMCID imcID,
      /*in*/  TNC_ConnectionID connectionID,
      /*in*/  TNC_AttributeID attributeID,
      /*in*/  TNC_UInt32 bufferLength,
      /*in*/  TNC_BufferReference buffer);
```

Description:

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

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

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

The IMC 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 3.6.8 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 TNCC MUST NOT modify the contents of the buffer or even access them after this function returns. Therefore, the TNCC SHOULD copy the attribute value to other storage before returning from this function.

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

If the IMC passes an attribute value that is not valid for the specified attribute, the TNCC MAY return the `TNC_RESULT_INVALID_PARAMETER` result code to indicate that the attribute value is not valid. The TNCC 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.

Some attributes have different values for different IMCs (e.g. the IMC Supports TNCS First attribute). These attributes are identified as such when they are defined. If a TNCC supports one of these attributes, the TNCC MUST maintain separate values for the attribute per IMC (and per connection if the attribute is a connection-specific attribute).

Input Parameter	**Description**
`imcID`	IMC ID assigned by TNCC
`connectionID`	Network connection ID for which an attribute value is to be set (or `TNC_CONNECTIONID_ANY` to set an attribute value for the TNCC 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_IMC_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

3.9.7 TNC_TNCC_ReserveAdditionalIMCID (OPTIONAL)

```
TNC_Result TNC_TNCC_ReserveAdditionalIMCID(
      /*in*/  TNC_IMCID imcID,
      /*out*/  TNC_UInt32 *pOutIMCID);
```

Description:

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

When the IF-TNCCS 2.0 [10] (and PB-TNC [17]) 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 IMCs and IMVs 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.

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

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

If the TNCC 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 IMC ID that has now been reserved for this IMC.

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

Input Parameter	**Description**
`imcID`	Primary IMC ID assigned by TNCC

Output Parameter	**Description**
`pOutIMCID`	Requested additional IMC ID

Result Code	**Condition**
`TNC_RESULT_SUCCESS`	Success
`TNC_RESULT_NOT_INITIALIZED`	`TNC_IMC_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-IMC 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-IMC SHOULD be implemented on specific platforms. Additional platform bindings will be defined later.

4.1 Microsoft Windows DLL Platform Binding

Microsoft Windows is a popular platform with many variations. This binding does not support 16-bit Windows (Windows 3.X and Windows for Workgroups). It does support Windows XP, Windows CE, Windows NT, Windows 98, Windows Me, Windows 95, 64-bit Windows, and all other currently known versions of Windows.

Implementations on one of these platforms SHOULD use this binding when possible for maximum compatibility with other IMCs and TNC Clients 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 IMCs

One factor in Windows' success has been the ease with which software can be installed and configured. However, this can also lead to security problems if untrusted users install unsafe software. We retain this ease of configuration while providing some protection against unsafe software. Use of the Trusted Platform Module will increase the protection against unsafe software configuration.

With the Microsoft Windows DLL platform binding, each IMC is implemented as a DLL. 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 that can only be changed by privileged users. The TNC Client gets the value of this key and loads the IMCs using the `LoadLibrary` system call. Then it uses the `GetProcAddress` function call to access the IMC's functions, as described in section 4.1.2. The TNCC MUST always call the `TNC_IMC_Initialize` function first. When it is done using an IMC, the TNC Client calls `TNC_IMC_Terminate` and then unloads the IMC DLL using the `FreeLibrary` system call. The TNCC SHOULD listen for changes to the well-known registry key so that it can load and unload IMCs dynamically. However, the TNCC SHOULD delay before making changes based on registry key changes since it's common for these changes to come in batches within a few seconds during an install process. And the TNCC MAY not listen for such changes at all.

Because the TNCC uses the `LoadLibrary` system call to load the IMC, DLLs implicitly referenced by the IMC 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 IMC does not contain the directory containing the IMC DLL. It only contains the directory containing the TNCC, system directories, etc. Two good solutions to this problem are to have the IMC explicitly load any unusual DLLs that it needs and use the `GetProcAddress` function call to access functions within those DLLs, or to have the IMC delay-load any unusual DLLs and write a delay load hook that searchs the IMC's paths for those libraries.

4.1.1.1 Use of COM Objects

The TNCC and IMCs 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 TNCC or IMCs 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 TNCC has initialized the COM library with the `COINIT_APARTMENTTHREADED` option, an IMC 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 TNCC and IMCs may need to initialize the COM library in mutually inconsistent ways, an IMC should not make any assumptions about the initialization parameters used by the TNCC or even whether the TNCC has initialized the COM library. Having an IMC invoke methods on COM objects in a thread shared with the TNCC could potentially cause problems. Since TNC client software is a multivendor environment where the TNCC, IMCs, and third party libraries used by IMCs 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, IMCs 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 and take care of properly initializing and uninitializing the COM library on those threads. If an IMC 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 IMC can ignore this section.

4.1.1.2 Security Issues

On some versions of Windows (including at least Windows 95, Windows 98, and Windows ME), there is no such thing as a privileged user. This means that any code executed with the privileges of any user can modify the registry key that lists the installed IMCs. However, this problem is not unique to the IF-IMC API. It's commonly known that running malicious code on such an operating system may result in complete compromise of the machine. The chances of such an attack can be reduced through best practices like well-patched software, strong host intrusion prevention, and antivirus protection. The TNC architecture supports and encourages such measures. IF-IMC helps ensure that these measures are in place. IF-PTS allows the TNCS to reliably and securely detect compromised machines through use of the TPM. But upgrading to a more secure version of Windows is also recommended.

As described in the Security Considerations section, loading an IMC DLL into the TNCC's address space can compromise the TNCC and other IMCs if the IMC DLL is later found to be untrustworthy. Also, an unstable IMC can crash the whole TNCC. One way to address this problem is to have the TNCC launch a new "child" process for each IMC, have the child process load the IMC DLL, and then have the TNCC communicate with the child processes carefully. If an IMC DLL crashes or is untrustworthy, the damage it can do is limited. The TNCC may use this approach but is not required to do so.

4.1.2 Dynamic Function Binding

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

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

IMC and TNCC 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 IMC and TNCC functions exposed through this API.

4.1.3 Threading

IMC DLLs are not required to be thread-safe. Therefore, the TNC Client MUST NOT call an IMC DLL from one thread when another TNC Client thread is in the middle of a call to the same IMC DLL. However, since more than one TNC Client may be running at once on a single machine (rare, but possible), any IMC DLL MUST be prepared to be loaded in multiple processes at once and to have these processes issue overlapping calls to the DLL.

The IMC DLL MAY create threads. The TNC Client MUST be thread-safe. This allows the IMC DLL to do work in background threads and call the TNC Client when it wants to request an Integrity Check Handshake retry (for instance).

All IMC DLL functions SHOULD return promptly. Otherwise, the TNC Client may get bogged down waiting for a response from the IMC. A long delay might frustrate users or exceed network timeouts (PDP, PEP or otherwise).

4.1.4 Platform-Specific Bindings for Basic Types

With the Microsoft Windows DLL platform binding, the basic data types defined in the IF-IMC 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-IMC 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-IMC 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_TNCC_BindFunction` to the right type and ensure that the compiler performs type checking on arguments.

```
typedef TNC_Result (*TNC_IMC_InitializePointer)(
    TNC_IMCID imcID,
    TNC_Version minVersion,
    TNC_Version maxVersion,
    TNC_Version *pOutActualVersion);
```

```
typedef TNC_Result (*TNC_IMC_NotifyConnectionChangePointer)(
    TNC_IMCID imcID,
    TNC_ConnectionID connectionID,
    TNC_ConnectionState newState);

typedef TNC_Result (*TNC_IMC_BeginHandshakePointer)(
    TNC_IMCID imcID,
    TNC_ConnectionID connectionID);

typedef TNC_Result (*TNC_IMC_ReceiveMessagePointer)(
    TNC_IMCID imcID,
    TNC_ConnectionID connectionID,
    TNC_BufferReference message,
    TNC_UInt32 messageLength,
    TNC_MessageType messageType);

typedef TNC_Result (*TNC_IMC_ReceiveMessageSOHPointer)(
    TNC_IMCID imcID,
    TNC_ConnectionID connectionID,
    TNC_BufferReference sohrReportEntry,
    TNC_UInt32 sohrRELength,
    TNC_MessageType systemHealthID);

typedef TNC_Result (*TNC_IMC_ReceiveMessageLongPointer)(
    TNC_IMCID imcID,
    TNC_ConnectionID connectionID,
    TNC_UInt32 messageFlags,
    TNC_BufferReference message,
    TNC_UInt32 messageLength,
    TNC_VendorID messageVendorID,
    TNC_MessageSubtype messageSubtype,
    TNC_UInt32 sourceIMVID,
    TNC_UInt32 destinationIMCID);

typedef TNC_Result (*TNC_IMC_BatchEndingPointer)(
    TNC_IMCID imcID,
    TNC_ConnectionID connectionID);

typedef TNC_Result (*TNC_IMC_TerminatePointer)(
    TNC_IMCID imcID);

typedef TNC_Result (*TNC_TNCC_ReportMessageTypesPointer)(
    TNC_IMCID imcID,
    TNC_MessageTypeList supportedTypes,
    TNC_UInt32 typeCount);

typedef TNC_Result (*TNC_TNCC_ReportMessageTypesLongPointer)(
    TNC_IMCID imcID,
    TNC_VendorIDList supportedVendorIDs,
    TNC_MessageSubtypeList supportedSubtypes,
    TNC_UInt32 typeCount);

typedef TNC_Result (*TNC_TNCC_SendMessagePointer)(
    TNC_IMCID imcID,
    TNC_ConnectionID connectionID,
    TNC_BufferReference message,
    TNC_UInt32 messageLength,
    TNC_MessageType messageType);

typedef TNC_Result (*TNC_TNCC_SendMessageSOHPointer)(
    TNC_IMCID imcID,
    TNC_ConnectionID connectionID,
    TNC_BufferReference sohReportEntry,
    TNC_UInt32 sohRELength);

typedef TNC_Result (*TNC_TNCC_SendMessageLongPointer)(
    TNC_IMCID imcID,
    TNC_ConnectionID connectionID,
    TNC_UInt32 messageFlags,
    TNC_BufferReference message,
    TNC_UInt32 messageLength,
    TNC_VendorID messageVendorID,
    TNC_MessageSubtype messageSubtype,
    TNC_UInt32 destinationIMVID);

typedef TNC_Result (*TNC_TNCC_RequestHandshakeRetryPointer)(
    TNC_IMCID imcID,
    TNC_ConnectionID connectionID,
    TNC_RetryReason reason);

typedef TNC_Result (*TNC_TNCC_GetAttributePointer)(
    TNC_IMCID imcID,
    TNC_ConnectionID connectionID,
    TNC_AttributeID attributeID,
    TNC_UInt32 bufferLength,
    TNC_BufferReference buffer,
    TNC_UInt32 *pOutValueLength);

typedef TNC_Result (*TNC_TNCC_SetAttributePointer)(
    TNC_IMCID imcID,
    TNC_ConnectionID connectionID,
    TNC_AttributeID attributeID,
    TNC_UInt32 bufferLength,
    TNC_BufferReference buffer);

typedef TNC_Result (*TNC_TNCC_ReserveAdditionalIMCIDPointer)(
    TNC_IMCID imcID,
    TNC_UInt32 *pOutIMCID);

typedef TNC_Result (*TNC_TNCC_BindFunctionPointer)(
    TNC_IMCID imcID,
    char *functionName,
    void **pOutfunctionPointer);

typedef TNC_Result (*TNC_IMC_ProvideBindFunctionPointer)(
    TNC_IMCID imcID,
    TNC_TNCC_BindFunctionPointer bindFunction);
```

4.1.7 Platform-Specific IMC Functions

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

4.1.7.1 TNC_IMC_ProvideBindFunction (MANDATORY)

```
TNC_Result TNC_IMC_ProvideBindFunction(
      /*in*/  TNC_IMCID imcID,
      /*in*/  TNC_TNCC_BindFunctionPointer bindFunction);
```

Description:

IMCs implementing the Microsoft Windows DLL platform binding MUST define this additional platform-specific function. The TNC Client MUST call the function immediately after calling `TNC_IMC_Initialize` to provide a pointer to the TNCC bind function. The IMC can then use the TNCC bind function to obtain pointers to any other TNCC functions.

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

Input Parameter	**Description**
`imcID`	IMC ID assigned by TNCC
`bindFunction`	Pointer to `TNC_TNCC_BindFunction`

Result Code	**Condition**
`TNC_RESULT_SUCCESS`	Success
`TNC_RESULT_NOT_INITIALIZED`	`TNC_IMC_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

4.1.8 Platform-Specific TNC Client Functions

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

4.1.8.1 TNC_TNCC_BindFunction (MANDATORY)

```
TNC_Result TNC_TNCC_BindFunction(
      /*in*/  TNC_IMCID imcID,
      /*in*/  char *functionName,
      /*out*/ void **pOutFunctionPointer);
```

Description:

TNC Clients implementing the Microsoft Windows DLL platform binding MUST define this additional platform-specific function. An IMC can use this function to obtain pointers to other TNCC functions. To obtain a pointer to a TNCC function, an IMC calls `TNC_TNCC_BindFunction`. The IMC obtains a pointer to `TNC_TNCC_BindFunction` from `TNC_IMC_ProvideBindFunction`.

The IMC MUST set the `imcID` parameter to the IMC ID value provided to `TNC_IMC_Initialize`. TNCCs MAY check if `imcID` matches the value previously passed to `TNC_IMC_Initialize` and return `TNC_RESULT_INVALID_PARAMETER` if not, but they are not required to make this check. The IMC MUST set the `functionName` parameter to a pointer to a C string identifying the function whose pointer is desired (i.e. `"TNC_TNCC_SendMessage"`). The IMC MUST set the `pOutFunctionPointer` parameter to a pointer to storage into which the desired function pointer will be stored. If the TNCC 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 IMC obtains a pointer to a particular function, the TNCC MUST always return the same function pointer value to that IMC for that function name. This requirement does not apply across IMC termination and reinitialization.

Input Parameter	**Description**
`imcID`	IMC ID assigned by TNCC
`functionName`	Name of function whose pointer is requested

Output Parameter	**Description**
`pOutFunctionPointer`	Requested function pointer

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

4.1.9 Well-known Registry Key

As discussed above, a well-known registry key is used by the TNCC to load IMCs. For Windows platforms, this key is defined within the HKEY_LOCAL_MACHINE hive as follows. The TNCC should also load IMCs from the equivalent path in HKEY_CURRENT_USER hive in addition to HKLM in order to support per-user software profiles.

- HKEY_LOCAL_MACHINE

  → Software

  → Trusted Computing Group

  → TNC

  → IMCs

  → [Human readable name of IMC], 0..n

Each IMC 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 IMC DLL to be loaded
- the optional value *"Description"* is a REG_SZ String which contains a vendor-specific human-readable description of the IMC DLL

The name and description are for ease of administration and may be ignored by the TNCC, 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 Clients and IMCs 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 Client vendor with vendor ID 2 could specify that any IMC can populate its key at install time with a value named 2_SupportPhone and that vendor's TNC Client can read this value and display it in the TNC Client's status panel next to the IMC name. The only requirement, as stated above, is that TNC Clients and IMCs 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 IMCs and TNC Clients 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 IMCs

With the UNIX/Linux Dynamic Linkage platform binding, each IMC is implemented as a dynamically loaded executable file (also known as a shared object or DLL). When the IMC 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 Client 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 IMC to be loaded, the TNC Client 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 Client uses the `dlsym` function call to access the IMC's functions, as described in section 4.1.2. The TNCC MUST always call the `TNC_IMC_Initialize` function first. When it is done using an IMC, the TNC Client calls `TNC_IMC_Terminate` and then unloads the IMC executable file using the `dlclose` system call.

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

As described in the Security Considerations section, loading an IMC into the TNCC's address space can compromise the TNCC and other IMCs if the IMC is later found to be untrustworthy. Also, an unstable IMC can crash the whole TNCC. One way to address this problem is to have the TNCC launch a new "child" process for each IMC, have the child process load the IMC, and then have the TNCC communicate with the child processes carefully. If an IMC crashes or is untrustworthy, the damage it can do is limited. The TNCC may use this approach but is not required to do so.

4.2.2 Dynamic Function Binding

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

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

IMC and TNCC 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 IMC and TNCC functions exposed through this API.

4.2.3 Format of `/etc/tnc_config`

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

The `/etc/tnc_config` file is a UTF-8 file. However, TNCCs are only required to support US-ASCII characters (a subset of UTF-8). If a TNCC encounters a character that is not US-ASCII and the TNCC can not process UTF-8 properly, the TNCC SHOULD indicate an error and not load the file at all. In fact, the TNCC 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 "IMC " (U+0049, U+004D, U+0043, U+0020) specifies an IMC that may be loaded. The next character MUST be U+0022 (QUOTATION MARK). This MUST be followed by a human-readable IMC name (potentially zero length) and another U+0022 character (QUOTATION MARK). Of course, the IMC 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 IMC executable file (up to but not including the U+000A that terminates the line). The path to the IMC executable file MUST NOT be a partial path.

The `/etc/tnc_config` file must not contain IMCs with the same human-readable name. A TNCC that encounters such a file SHOULD indicate the error and MAY not load the file at all. It MAY also change the IMC 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 TNCC vendor with vendor ID 2 could specify that any IMC can add a line at install time that begins with 2_SupportPhoneIMC, then the IMC's human-readable name and the IMC vendor's support telephone number. The defining vendor's TNCC (or any other TNCC) can read this phone number and display it in the TNCC's status panel next to the IMC name.

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

A line that does not match the comment, empty, imc, or vendor productions below SHOULD be ignored by a TNCC and IMCs 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 java-`imc`, `imv`, and `java-imv` productions are ignored for the purposes of the Linux/UNIX Platform Binding for IF-IMC. Note also that the `other` production is only employed if no other production matches a line.

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

```
# Simple TNC config file

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

4.2.4 Threading

IMC executable files are not required to be thread-safe. Therefore, the TNC Client MUST NOT call an IMC from one thread when another TNC Client thread is in the middle of a call to the same IMC. However, since more than one TNC Client may be running at once on a single machine (rare, but possible), any IMC MUST be prepared to be loaded in multiple processes at once and to have these processes issue overlapping calls to the IMC.

The IMC MAY create threads. The TNC Client MUST be thread-safe. This allows the IMC to do work in background threads and to call the TNC Client when it wants to request an Integrity Check Handshake retry (for instance). Both the IMC and the TNC Client MUST use POSIX threads (pthreads) for threading and synchronization to ensure compatibility.

All IMC functions SHOULD return promptly (preferably, within 100 ms or less). Otherwise, the TNC Client may get bogged down waiting for a response from the IMC.

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-IMC 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-IMC 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 platform binding for the IF-IMC 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_TNCC_BindFunction` to the right type and ensure that the compiler performs type checking on arguments.

```
typedef TNC_Result (*TNC_IMC_InitializePointer)(
    TNC_IMCID imcID,
    TNC_Version minVersion,
    TNC_Version maxVersion,
    TNC_Version *pOutActualVersion);

typedef TNC_Result (*TNC_IMC_NotifyConnectionChangePointer)(
    TNC_IMCID imcID,
    TNC_ConnectionID connectionID,
    TNC_ConnectionState newState);

typedef TNC_Result (*TNC_IMC_BeginHandshakePointer)(
    TNC_IMCID imcID,
    TNC_ConnectionID connectionID);

typedef TNC_Result (*TNC_IMC_ReceiveMessagePointer)(
    TNC_IMCID imcID,
    TNC_ConnectionID connectionID,
    TNC_BufferReference message,
    TNC_UInt32 messageLength,
    TNC_MessageType messageType);

typedef TNC_Result (*TNC_IMC_ReceiveMessageSOHPointer)(
    TNC_IMCID imcID,
    TNC_ConnectionID connectionID,
    TNC_BufferReference sohrReportEntry,
    TNC_UInt32 sohrRELength,
    TNC_MessageType systemHealthID);

typedef TNC_Result (*TNC_IMC_ReceiveMessageLongPointer)(
    TNC_IMCID imcID,
    TNC_ConnectionID connectionID,
    TNC_UInt32 messageFlags,
    TNC_BufferReference message,
    TNC_UInt32 messageLength,
    TNC_VendorID messageVendorID,
    TNC_MessageSubtype messageSubtype,
    TNC_UInt32 sourceIMVID,
    TNC_UInt32 destinationIMCID);

typedef TNC_Result (*TNC_IMC_BatchEndingPointer)(
    TNC_IMCID imcID,
    TNC_ConnectionID connectionID);

typedef TNC_Result (*TNC_IMC_TerminatePointer)(
    TNC_IMCID imcID);

typedef TNC_Result (*TNC_TNCC_ReportMessageTypesPointer)(
    TNC_IMCID imcID,
    TNC_MessageTypeList supportedTypes,
    TNC_UInt32 typeCount);

typedef TNC_Result (*TNC_TNCC_ReportMessageTypesLongPointer)(
    TNC_IMCID imcID,
    TNC_VendorIDList supportedVendorIDs,
    TNC_MessageSubtypeList supportedSubtypes,
    TNC_UInt32 typeCount);

typedef TNC_Result (*TNC_TNCC_SendMessagePointer)(
    TNC_IMCID imcID,
    TNC_ConnectionID connectionID,
    TNC_BufferReference message,
    TNC_UInt32 messageLength,
    TNC_MessageType messageType);

typedef TNC_Result (*TNC_TNCC_SendMessageSOHPointer)(
    TNC_IMCID imcID,
    TNC_ConnectionID connectionID,
    TNC_BufferReference sohReportEntry,
    TNC_UInt32 sohRELength);

typedef TNC_Result (*TNC_TNCC_SendMessageLongPointer)(
    TNC_IMCID imcID,
    TNC_ConnectionID connectionID,
    TNC_UInt32 messageFlags,
    TNC_BufferReference message,
    TNC_UInt32 messageLength,
    TNC_VendorID messageVendorID,
    TNC_MessageSubtype messageSubtype,
    TNC_UInt32 destinationIMVID);

typedef TNC_Result (*TNC_TNCC_RequestHandshakeRetryPointer)(
    TNC_IMCID imcID,
    TNC_ConnectionID connectionID,
    TNC_RetryReason reason);

typedef TNC_Result (*TNC_TNCC_GetAttributePointer)(
    TNC_IMCID imcID,
    TNC_ConnectionID connectionID,
    TNC_AttributeID attributeID,
    TNC_UInt32 bufferLength,
    TNC_BufferReference buffer,
    TNC_UInt32 *pOutValueLength);

typedef TNC_Result (*TNC_TNCC_SetAttributePointer)(
    TNC_IMCID imcID,
    TNC_ConnectionID connectionID,
    TNC_AttributeID attributeID,
    TNC_UInt32 bufferLength,
    TNC_BufferReference buffer);

typedef TNC_Result (*TNC_TNCC_ReserveAdditionalIMCIDPointer)(
    TNC_IMCID imcID,
    TNC_UInt32 *pOutIMCID);

typedef TNC_Result (*TNC_TNCC_BindFunctionPointer)(
    TNC_IMCID imcID,
    char *functionName,
    void **pOutfunctionPointer);

typedef TNC_Result (*TNC_IMC_ProvideBindFunctionPointer)(
    TNC_IMCID imcID,
    TNC_TNCC_BindFunctionPointer bindFunction);
```

4.2.8 Platform-Specific IMC Functions

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

4.2.8.1 TNC_IMC_ProvideBindFunction (MANDATORY)

```
TNC_Result TNC_IMC_ProvideBindFunction(
      /*in*/  TNC_IMCID imcID,
      /*in*/  TNC_TNCC_BindFunctionPointer bindFunction);
```

Description:

IMCs implementing the UNIX/Linux Dynamic Linkage platform binding MUST define this additional platform-specific function. The TNC Client MUST call the function immediately after calling `TNC_IMC_Initialize` to provide a pointer to the TNCC bind function. The IMC can then use the TNCC bind function to obtain pointers to any other TNCC functions.

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

Input Parameter	**Description**
`imcID`	IMC ID assigned by TNCC
`bindFunction`	Pointer to `TNC_TNCC_BindFunction`

Result Code	**Condition**
`TNC_RESULT_SUCCESS`	Success
`TNC_RESULT_NOT_INITIALIZED`	`TNC_IMC_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

4.2.9 Platform-Specific TNC Client Functions

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

4.2.9.1 TNC_TNCC_BindFunction (MANDATORY)

```
TNC_Result TNC_TNCC_BindFunction(
      /*in*/  TNC_IMCID imcID,
      /*in*/  char *functionName,
      /*out*/ void **pOutFunctionPointer);
```

Description:

TNC Clients implementing the UNIX/Linux Dynamic Linkage platform binding MUST define this additional platform-specific function. An IMC can use this function to obtain pointers to other TNCC functions. To obtain a pointer to a TNCC function, an IMC calls TNC_TNCC_BindFunction. The IMC obtains a pointer to TNC_TNCC_BindFunction from TNC_IMC_ProvideBindFunction.

The IMC MUST set the imcID parameter to the IMC ID value provided to TNC_IMC_Initialize. TNCCs MAY check if imcID matches the value previously passed to TNC_IMC_Initialize and return TNC_RESULT_INVALID_PARAMETER if not, but they are not required to make this check. The IMC MUST set the functionName parameter to a pointer to a C string identifying the function whose pointer is desired (i.e. "TNC_TNCC_SendMessage"). The IMC MUST set the pOutFunctionPointer parameter to a pointer to storage into which the desired function pointer will be stored. If the TNCC 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**
imcID	IMC ID assigned by TNCC
functionName	Name of function whose pointer is requested

Output Parameter	**Description**
pOutFunctionPointer	Requested function pointer

Result Code	**Condition**
TNC_RESULT_SUCCESS	Success
TNC_RESULT_OTHER	Unspecified non-fatal error
TNC_RESULT_FATAL	Unspecified fatal error
TNC_RESULT_INVALID_PARAMETER	Invalid function parameter
Other result 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 sandboxing features that make it a popular choice for running dynamically downloaded and perhaps only partially trusted code. There is a desire to support IF-IMC on the Java Platform, especially to support dynamic download of IMCs and perhaps of the TNCC. Therefore, the Java Platform Binding for IF-IMC has been developed.

At this time, the only versions of the Java Platform that are supported with the Java Platform Binding for IF-IMC are the Java 2 Platform Standard Edition versions 1.4.2 and later. Other Java Platform versions may be supported at a later time. Implementations of the IF-IMC 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-IMC 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-IMC Abstract API: `IMC`, `TNCC`, and `IMCConnection`. The functions described in the IF-IMC 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 IMCs that implement the Java Platform Binding for IF-IMC MUST implement the `IMC` interface. All TNCCs that implement the Java Platform Binding for IF-IMC MUST implement the `TNCC` interface and provide objects that implement the `IMCConnection` interface as needed. The TNCC MUST also define all the interfaces in the IF-IMC 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-IMC Abstract API since exceptions must be explicitly ignored while result codes are ignored by default. Therefore, the Java Platform Binding for IF-IMC defines a `TNCException` class that wraps the result codes defined in the IF-IMC Abstract API. This class MUST be defined by all TNCCs.

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-IMC, where it may be desirable to dynamically download a TNCC and/or IMCs and limit what this partially trusted code can do.

The Java Platform Binding for IF-IMC does not define any new Permissions. All IMCs loaded by the TNCC are assumed to be trusted to call any TNCC methods and vice versa. However, this does not mean that the TNCC and IMCs 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 TNCC may have more or fewer privileges than an IMC if the TNCC and IMC were loaded from different CodeSources, a Java TNCC and IMC cannot simply trust each other.

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

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

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

Of course, a TNCC or IMC with limited privileges have somewhat limited utility. For example, a TNCC needs at least enough privileges to load IMCs and communicate with the network. An IMC needs enough privileges to check endpoint integrity. If the IMC is expected to perform remediation, it will probably need additional privileges to do that (updating files, changing settings, etc.).

4.3.4 Finding, Loading, and Unloading IMCs

With the Java platform binding, each IMC is implemented as a jar file. When the IMC is installed and is intended to be usable by any TNCC 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-IMC 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 Client that wishes to load a Java IMC SHOULD open the `tnc_config` file on the system, read the JAVA-IMC entries in the file, and determine which of them should be loaded (using optional local configuration or any other algorithm). For each IMC to be loaded, the TNC Client SHOULD create a new instance of the IMC class, using the full path of the jar file and the class name for the IMC class as specified in the `tnc_config` file to load the class and call the noargs constructor for that class. When loading an IMC class in this manner, the TNCC MUST use a class loader that will determine and assign the appropriate permissions (such as the URLClassLoader). The TNCC SHOULD check the `tnc_config` file for changes and load or unload IMCs as needed to match the latest list.

The TNCC MUST always call the IMC's `initialize` method first. When it is finished with an IMC, the TNC Client MUST call the IMC's `terminate` method. The unloading of the class will be handled by the JVM.

As described in the Security Considerations section, loading an IMC into the same JVM as the TNCC can compromise the TNCC and other IMCs if the IMC is later found to be untrustworthy. Also, an unstable IMC can crash the whole TNCC. 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 IMC are minimal.

4.3.5 Dynamic Function Binding

The Java Platform Binding for IF-IMC does support dynamic function binding. Thus, to allow a TNCC or IMC 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 Client to determine whether an optional IMC method is implemented, the TNC Client should make a call to the method. If the method is not implemented, an `UnsupportedOperationException` is thrown. Similarly, an IMC can call an optional TNCC method, and if the method is not implemented by the TNCC then an `UnsupportedOperationException` will be thrown. However, these techniques only apply to methods that were defined in IF-IMC 1.1 (where the Java Platform Binding for IF-IMC was originally defined).

Extensions to the IF-IMC API beyond the methods defined in IF-IMC 1.1 (new features added to the standard IF-IMC API and vendor extensions to that API) are handled using interfaces. To define an extension to the IF-IMC 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 IMC or TNCC 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-IMC 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 IMCs available for TNCCs to load. TNCCs are not required to load these IMCs. A TNCC may be configured to ignore this file, load any subset of the IMCs listed here, load a superset of those IMCs, or load the IMCs in the list. This provides a simple, standard way for the list of IMCs to be specified but allows TNCCs to be configured to only load a particular set of trusted IMCs.

The `tnc_config` file is a UTF-8 file. However, TNCCs are only required to support US-ASCII characters (a subset of UTF-8). If a TNCC encounters a character that is not US-ASCII and the TNCC can not process UTF-8 properly, the TNCC SHOULD indicate an error and not load the file at all. In fact, the TNCC 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-IMC ” (U+004A, U+0041, U+0056, U+0041, U+002D, U+0049, U+004D, U+0043, U+0020) specifies a Java IMC (an IMC 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 IMC name (potentially zero length) and another U+0022 character (QUOTATION MARK). Of course, the human-readable IMC 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 IMC class (which MUST NOT include a space), followed by a space (U+0020). After this space MUST come the full path of the IMC jar file (which runs up to but does not include the

U+000A that terminates the line). The path to the IMC 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 IMC with the same human-readable name. A TNCC that encounters such a file SHOULD indicate the error and MAY not load the file at all. It MAY also change the IMC names to make them unique. Identical class names and full paths are permitted but the TNCC 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 TNCC vendor with vendor ID 2 could specify that any IMC can add a line at install time that begins with 2_SupportPhoneIMC, then the IMC’s human-readable name and the IMC vendor’s support telephone number. The defining vendor’s TNCC (or any other TNCC) can read this phone number and display it in the TNCC’s status panel next to the IMC name.

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

A line that does not match the comment, empty, java-imc, or vendor productions below SHOULD be ignored by a TNCC and IMCs that are using the Java 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`, `imv`, and `java-imv` productions are ignored for the purposes of the Java Platform Binding for IF-IMC. Note also that the `other` production is only employed if no other production matches a line.

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

```
# Simple Java IMC config file

JAVA-IMC "Example IMC" com.example.ExampleIMC /usr/bin/example_imc.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. IMCs and TNCCs 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 IMC or TNCC 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-IMC, IMCs are not required to be thread-safe. Therefore, the TNC Client MUST NOT call an IMC from one thread when another TNC Client thread is in the middle of a call to the same IMC. However, since more than one TNC Client may be running at once on a single machine (rare, but possible), any IMC MUST be prepared to be loaded in multiple processes at once and to have these processes issue overlapping calls to the IMC.

An IMC MAY create threads. The TNC Client MUST be thread-safe. This allows the IMC to do work in background threads and to call the TNC Client when it wants to request an Integrity Check Handshake retry (for instance).

All IMC functions SHOULD return promptly (preferably, within 100 ms or less). Otherwise, the TNC Client may get bogged down waiting for a response from the IMC.

4.3.9 Attributes

Instead of representing attribute values with a byte array, the Java Platform Binding for IF-IMC uses objects. For each attribute ID defined in section 3.6.8, this section explains the object used by the Java Platform Binding for IF-IMC to represent values for that attribute.

4.3.9.1 Maximum Round Trips Attribute

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

With the Java Platform Binding for IF-IMC, 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.

4.3.9.2 Maximum Message Size Attribute

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

With the Java Platform Binding for IF-IMC, 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.

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

The DHPN attribute allows a IMCConnection that supports Diffie-Hellman Pre-Negotiation (as described in IF-T for Tunneled EAP Methods) to provide to a PTS-IMC or other IMCs a Unique-Value-1, so that this value can be used in the TPM_Quote operation.

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

4.3.9.4 Has Long Types Attribute

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

With the Java Platform Binding for IF-IMC, attribute values for the Has Long Types attribute MUST be 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.

4.3.9.5 Has Exclusive Attribute

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

With the Java Platform Binding for IF-IMC, 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.

4.3.9.6 Has SOH Attribute

The Has SOH attribute allows a TNCC to indicate to the IMCs that a particular connection supports SOH functions like `TNC_TNCC_SendMessageSOH`.

With the Java Platform Binding for IF-IMC, 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.

4.3.9.7 SOHR Attribute

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

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

4.3.9.8 SSOHR Attribute

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

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

4.3.9.9 IF-TNCCS Protocol Attribute

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

With the Java Platform Binding for IF-IMC, 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-IMC uses essentially the same attribute value as the abstract binding but translates it into the types and classes native to the Java platform.

4.3.9.10 IF-TNCCS Version Attribute

The IF-TNCCS Version attribute allows IMCs 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-IMC, 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-IMC 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 IF-T Protocol Attribute

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

With the Java Platform Binding for IF-IMC, 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-IMC 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-T Version Attribute

The IF-T Version attribute allows IMCs 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-IMC, 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-IMC 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 IMC Supports TNCS First Attribute

The IMC Supports TNCS First attribute allows an IMC to indicate to the TNCC that the IMC supports having the TNCS send the first batch in an exchange.

With the Java Platform Binding for IF-IMC, attribute values for the IMC Supports TNCS First attribute are represented as a java.lang.Boolean indicating whether the IMC supports having the TNCS send the first batch in an exchange. A value of java.lang.Boolean.TRUE indicates that the IMC supports having the TNCS send the first batch, and a value of java.lang.Boolean.FALSE indicates that the IMC does not support having the TNCS send the first batch.

4.3.9.14 Primary IMC ID Attribute

The Primary IMC ID attribute indicates the unique identifier of the IMC assigned by the TNCC when the TNCC loaded this IMC.

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

4.3.9.15 TLS-Unique Attribute

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

With the Java Platform Binding for IF-IMC, 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-IMC 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-IMC 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-IMC specification defines a set of interfaces that must be implemented by TNC Clients and IMCs that implement the Java Platform Binding for IF-IMC. These interfaces allow the TNC Clients and IMCs 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-IMC, as they might be implemented by a typical TNC Client and IMC.

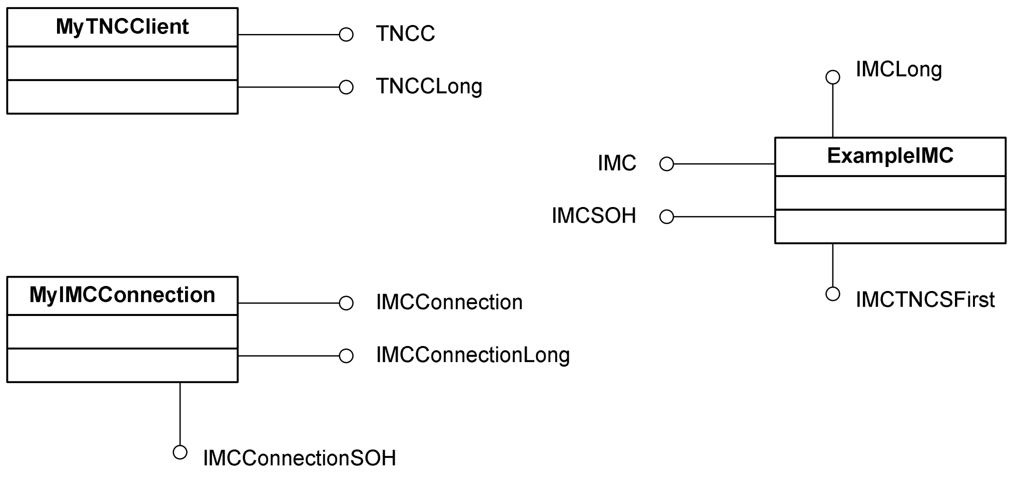


Figure 1: Typical Interface and Class Relationships

In this example, MyTNCClient implements the `TNCC` and `TNCCLong` interfaces because it not only supports the basic methods required of all TNC Clients but also supports IF-TNCCS 2.0 and therefore supports the extra methods defined in `TNCCLong`. The implementer of MyTNCClient has also created a separate class named MyIMCConnection. The MyIMCConnection class implements `IMCConnection` (including the basic methods used by an IMC to send messages and perform other operations on a connection) and also `IMCConnectionLong` and `IMCConnectionSOH`, because this class supports the extra methods included in those interfaces. Finally, ExampleIMC was created by another implementer. It supports not only the basic `IMC` 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 MyTNCClient examined the `tnc_config` file, it used the fully qualified class name and JAR file path in that file to create an instance of ExampleIMC. Once this has been done, MyTNCClient can discover which interfaces are supported by this IMC by using the `instanceof` operator to check whether the ExampleIMC object, which must implement the `IMV` interface, also implements the `IMCLong` interface. If so, MyTNCClient can cast the ExampleIMC object to the `IMCLong` type and use the methods in that interface. The same mechanism is also used by ExampleIMC to discover that the `TNCCLong` interface is implemented by MyTNCClient. This extensibility mechanism will be used in future versions of IF-IMC, 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-IMC API.

4.3.12.1 TNCException Class (TNCException.java)

org.trustedcomputinggroup.tnc

Class TNCException

```
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 a TNCException.

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.

`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 Interface (TNCConstants.java)

org.trustedcomputinggroup.tnc

Interface TNCConstants

`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.

4.3.12.3 TNCC Interface (TNCC.java)

org.trustedcomputinggroup.tnc.ifimc

Interface TNCC

```
public interface TNCC
```

These methods are implemented by the TNC Client and called by the IMC.

Method Detail

```
void reportMessageTypes(IMCIMC imc,
                        long[] supportedTypes)
                        throws TNCException
```

A call to this method is used to inform a TNCC about the set of message types that the IMC wishes to receive. Often, the IMC will call this method from the IMC'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 Clients MUST implement this method. The TNC Client MUST NOT ever modify the list of message types and MUST NOT access this list after TNCC reportMessageTypes() method has returned. Generally, the TNC Client will copy the contents of this list before returning from this method. TNC Clients MUST support any message type.

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

If an IMC 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 IMC. 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, except for messages marked for exclusive delivery to another IMC. If an IMC calls the TNCC'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:
`imc` - the IMC reporting its message types
`supportedTypes` - the message types the IMC wishes to receive
Throws:
`TNCException` - if a TNC error occurs

```
void requestHandshakeRetry(IMC imc,
                           long reason)
                           throws TNCException
```

IMCs can call this method to ask a TNCC to retry an Integrity Check Handshake for all current network connections. The IMC MUST pass itself as the imc parameter and one of the handshake retry reasons in IMCConnection as the reason parameter.

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

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

Parameters:
`imc` - an IMC requesting a retry handshake
`reason` - the reason for the handshake request
Throws:
`TNCException` - if a TNC error occurs

4.3.12.4 TNCCLong Interface (TNCCLong.java)

org.trustedcomputinggroup.tnc.ifimc

Interface TNCCLong

`public interface` **`TNCCLong`**

This interface can be implemented by an object that already supports the TNCC interface, to indicate that this object is a TNCC that supports reportMethodTypesLong. 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(IMC imc,
                            long[] supportedVendorIDs,
                            long[] supportedSubtypes)
                            throws TNCException
```

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

Often, the IMC will call this method from the IMC'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 IMC is able to receive.

This method is optional and is not supported by all TNCCs. IMCs can easily determine which TNCCs support this method by checking if they implement the TNCCLong interface. IMCs should recognize that many TNCCs do not support long types and therefore will not support this method. In those cases, IMCs should be prepared to use the reportMessageTypes function. Simple IMCs that do not receive messages need not call this function.

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

If an IMC 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 IMC. 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, except for messages marked for exclusive delivery to another IMC. If an IMC calls the TNCC'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:
`imc` - the IMC reporting its message types
`supportedVendorIDs` - the list of Vendor IDs the IMC wishes to receive
`supportedSubtypes` - the list of message subtypes the IMC wishes to receive
Throws:
`TNCException` - if a TNC error occurs

```
long reserveAdditionalIMCID(IMC imc)
                            throws TNCException
```

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

When the IF-TNCCS 2.0 [10] (and PB-TNC [17]) 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 IMC can call this function as many times it wishes. The TNCC SHOULD return a unique value every time. However, the TNCC may have a maximum number of additional IMC IDs that it supports. In that case the TNCC SHOULD return a `TNC_RESULT_OTHER` error if an IMC attempts to exceed this maximum. The TNCC is not required to reserve IMC IDs in a specific order.

Parameters:
`imc` - the IMC requesting additional IMC IDs

Throws:
TNCException - if a TNC error occurs

4.3.12.5 AttributeSupport Interface (AttributeSupport.java)

org.trustedcomputinggroup.tnc.ifimc

Interface AttributeSupport

`public interface` **`AttributeSupport`**
These methods can be implemented by an object in the IMC package to get or set attributes. This interface also includes a collection of common IMCConnection attributes to be used by the IMC package. To check whether an object in the IMC package implements this interface, use the `instanceof` operator. If so, cast the object to the interface type and use the new methods and fields.

Field Detail

`static final long` **`TNC_ATTRIBUTEID_PREFERRED_LANGUAGE`**
Preferred human-readable language for a TNCS (type String, may get from an IMCConnection)

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

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

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

`static final long` **`TNC_ATTRIBUTEID_SOHR`**
Contents of SOHR (type byte [], may get from an IMCConnection)

`static final long` **`TNC_ATTRIBUTEID_SSOHR`**
Contents of SSOHR (type byte [], may get from an IMCConnection)

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

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

`static final long` **`TNC_ATTRIBUTEID_IFT_PROTOCOL`**
IF-T Protocol Name (type String, may get from an IMCConnection)

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

`static final long` **`TNC_ATTRIBUTEID_IMC_SPTS_TNCS1`**

Flag to indicate if IMC supports TNCS sending first message (type boolean, may get from an IMC)

`static final long` **`TNC_ATTRIBUTEID_PRIMARY_IMC_ID`**

IMC identifier assigned by the TNCC when the TNCC loaded this IMC

`static final long` **`TNC_ATTRIBUTEID_TLS_UNIQUE`**

TLS-Unique value provided by the underlying protocol (type byte[], may get from an IMCConnection)

Method Detail

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

A call to this method gets the value of the attribute identified by `attributeID` for an object in the IMC package which implements this interface.

This function is optional. The objects in the IMC package are not required to implement this function. If this function is not implemented, it MUST throw an UnsupportedOperationException. Objects in the IMC package calling this function MUST work properly if this function is not implemented.

Objects of the IMC package MUST pass a standard or vendor-specific attribute ID as the attributeID parameter when calling this method. If the called object does not recognize the attribute ID, it SHOULD throw a TNCException with the TNC_RESULT_INVALID_PARAMETER result code. If the called object recognizes the attribute ID but does not have an attribute value for the requested attribute ID, 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 calling object must cast this `Object` to the class documented in the description of that specific attribute to get the desired value. All `Object`s 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 IMC calls this method to set the value of the attribute identified by `attributeID` for an object in the IMC package which implements this interface.

This function is optional. The objects in the IMC package are not required to implement this function. If this function is not implemented, it MUST throw an UnsupportedOperationException. Objects in the IMC package calling this function MUST work properly if this function is not implemented.

Objects of the IMC package MUST pass a standard or vendor-specific attribute ID as the attributeID parameter when calling this method. If the called object does not recognize the attribute ID, it SHOULD throw a TNCException with the TNC_RESULT_INVALID_PARAMETER result code. If the called object recognizes the attribute ID but does not have an attribute value for the requested attribute ID, it SHOULD also throw a TNCException with the TNC_RESULT_INVALID_PARAMETER result code.

For the `attributeValue` parameter, the caller 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 called object has any uncertainty about the immutability of the `Object`, the called object SHOULD copy the `Object`. The called object 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`

4.3.12.6 IMC Interface (IMC.java)

org.trustedcomputinggroup.tnc.ifimc

Interface IMC

```
public interface IMC
```

An Integrity Measurement Collector (IMC). These methods are implemented by the IMC and called by the TNC Client.

Method Detail

```
void initialize(TNCC tncc)
                throws TNCException
```

Initializes the IMC. The TNC Client supplies itself as a parameter so the IMC can call the TNCC as needed.

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

Parameters:
`tncc` - the TNC Client
Throws:
`TNCException` - if a TNC error occurs

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

Closes down the IMC. The TNC Client calls this method to close down the IMC when all work is complete or the IMC throws an TNC_RESULT_FATAL exception. Once a call to an IMC's terminate method is made, the TNC Client MUST NOT call the IMC except to call the IMC's initialize method (which may not succeed if the IMC cannot reinitialize itself). Even if the IMC throws an exception from this method, the TNC Client MAY continue with its unload or shutdown procedure.

Throws:

`TNCException` - if a TNC error occurs

```
void notifyConnectionChange(IMCConnection c,
                            long newState)
                            throws TNCException
```

Informs the IMC that the state of the network connection identified by connection parameter has changed to a new state. All the possible values of the new state for this version of the IF-IMC API are identified in the TNCConstants class. The TNCC MUST NOT use any other values with this version of IF-IMC.

IMCs that want to track the state of network connections or maintain per-connection data structures SHOULD implement this method. If an IMC chooses to not implement this method it MUST throw an UnsupportedOperationException.

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

If the state is TNC_CONNECTION_STATE_ACCESS_ALLOWED or TNC_CONNECTION_STATE_ACCESS_ISOLATED, the IMC SHOULD proceed with any remediation instructions received during the Integrity Check Handshake. However, the IMC SHOULD be prepared for delays in network access or even complete denial of network access, even in these cases. Network access will often be delayed for a few seconds while an IP address is acquired. And network access may be denied if the NAA overrides the TNCS Action Recommendation reflected in the new state value.

If the state is TNC_CONNECTION_STATE_ACCESS_NONE, the IMC MAY discard any remediation instructions received during the Integrity Check Handshake or it MAY follow them if possible.

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 IMC SHOULD discard any state pertaining to this network connection and MUST NOT pass this network connection ID to the TNC Client after this method returns (unless the TNCC later creates another network connection with the same network connection ID).

Parameters:

`c` - the IMC connection object

`newState` - new network connection state

Throws:

`TNCException` - if a TNC error occurs

```
void beginHandshake(IMCConnection c)
                    throws TNCException
```

The TNC Client calls this method to indicate that an Integrity Check Handshake is beginning and solicit messages from IMCs for the first batch. The IMC 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.

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

All IMCs MUST implement this method.

Parameters:
`c` - the IMC connection object
Throws:
`TNCException` - if a TNC error occurs

```
void receiveMessage(IMCConnection c,
                    long messageType,
                    byte[] message)
                    throws TNCException
```

The TNC Client calls this method to deliver a message to the IMC. 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 IMV (or a TNCS or other party acting as an IMV).

If IF-TNCCS-SOH is used for a connection, and the IMC and TNCC implement receiveMessageSOH(), the TNCC SHOULD use that method to deliver the contents of SOHRReportEntries instead of using receiveMessage. However, if IF-TNCCS-SOH is used for a connection, but either the IMC or the TNCC does not implement receiveMessageSOH, receiveMessage SHOULD be used instead. For each SOHResponseEntry, the value contained in the first System-Health-ID attribute should be compared to the messageType values previously supplied in the IMC's most recent call to reportMessageTypes or reportMessageTypesLong. The SOHResponseEntry should be delivered to each IMC that has a match. If an IMC that does not support receiveMessageSOH (or when the TNCC does not support that method), receiveMessage SHOULD be employed. In that case, the TNCC 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 SOHResponseEntry MUST NOT be delivered to this IMC. The TNCC MUST pass, in the messageType parameter, the value contained in the first System-Health-ID attribute.

The IMC 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 IMC methods, the IMC SHOULD NOT wait a long time (definitely not more than a second) before returning from the IMC 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 IMC should implement this method if it wants to receive messages. Simple IMCs that only send messages need not implement this method but MUST at a minimum throw an UnsupportedOperationException. The IMC MUST NOT ever modify the buffer contents and MUST NOT access the buffer after the IMC receiveMessage() method has returned. If the IMC wants to retain the message, it should copy it before returning from the IMC receiveMessage() method.

The message parameter may be null to represent an empty message. In the messageType parameter, the TNCC MUST pass the type of the message. This value MUST match one of the TNC_MessageType values previously supplied by the IMC to the TNCC in the IMC's most recent call to the TNCC's reportMessageTypes() method. IMCs MAY check these parameters to make sure they are valid and throw an exception if not, but IMCs are not required to make these checks.

Parameters:
`c` - the IMC connection object
`messageType` - the type of message to be delivered
`message` - the message to be delivered
Throws:
`TNCException` - if a TNC error occurs

```
void batchEnding(IMCConnection c)
                 throws TNCException
```

The TNC Client calls this method to notify IMCs that all IMV messages received in a batch have been delivered and this is the IMC's last chance to send a message in the batch of IMC messages currently being collected. An IMC MAY implement this method if it wants to perform some actions after all the IMV messages received during a batch have been delivered (using the IMC receiveMessage(), receiveMessageLong(), or receiveMessageSOH() methods). This is especially useful for IMCs that have included a wildcard in the list of message types reported using the TNCC reportMessageTypes() or reportMessageTypesLong() methods. If an IMC chooses to not implement this method it MUST throw an UnsupportedOperationException.

An IMC MAY call the IMCConnection's sendMessage, sendMessageSOH, or sendMessageLong methods from this method. As with all IMC methods, the IMC 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` - the IMC connection object
Throws:
`TNCException` - if a TNC error occurs

4.3.12.7 IMCSOH Interface (IMCSOH.java)

org.trustedcomputinggroup.tnc.ifimc

Interface IMCSOH

```
public interface IMCSOH
```

This interface can be implemented by an object that already supports the IMC interface to indicate that this object is an IMC 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(IMCConnection c,
                       long systemHealthID,
                       byte[] sohrReportEntry)
                       throws TNCException
```

The TNC Client calls this method (if supported by the TNCC and implemented by an IMC) to deliver an SOHRReportEntry message to the IMC. This allows an IMC to receive the entire SOHRReportEntry instead of just the contents of the Vendor-Specific attribute, as would be the case if the receiveMessage method was used.

A TNCC that supports IF-TNCCS-SOH SHOULD support this method, and a TNCC that supports this function SHOULD call this method for a particular IMC and connection instead of calling receiveMessage, if the IMC implements this method and the connection uses IF-TNCCS-SOH. A TNCC MUST NOT call this method for a particular IMC and connection if the IMC does not implement this function, or the connection does not use IF-TNCCS-SOH. A TNCC will often find that some IMCs loaded by that TNCC implement this function and some do not. In that case, when the TNCC is handling a connection that uses IF-TNCCS-SOH, the TNCC would call receiveMessageSOH for the IMCs that implement this method and receiveMessage for the IMCs that do not implement receiveMessageSOH. A TNCC MUST NOT call both receiveMessage and receiveMessageSOH for a single SOHRReportEntry for a single IMC.

IMCs are not required to implement this method, but any IMC that implements the IMCSOH interface MUST implement this method. An IMC should implement receiveMessageSOH if it wants to receive the entire SOHRReportEntry instead of just the Vendor-Specific attribute when IF-TNCCS-SOH is used. However, IMCs should recognize that many TNCCs do not support IF-TNCCS-SOH and some TNCCs that do support IF-TNCCS-SOH will not support this function. Therefore, IMCs should be prepared to receive messages via receiveMessage in some cases when IF-TNCC-SOH is used. Simple IMCs that only send messages need not implement this method. IMC implementers may find that implementing receiveMessageSOH is more complex than implementing receiveMessage, since receiveMessageSOH must parse the SOHRReportEntry, while receiveMessage only needs to parse the content of the Vendor-Specific attribute within the SOHRReportEntry. For many IMCs, the content of the Vendor-Specific attribute is all that they need. Therefore, IMCs are not required to implement receiveMessageSOH.

The content of the SOHRReportEntry is contained in the buffer referenced by sohrReportEntry. The length of this array MUST NOT be zero, and the parameter MUST NOT be null, since zero-length SOHRReportEntries are not permitted by the IF-TNCCS-SOH protocol. The content of the first System-Health-ID attribute in the SOHRReportEntry is indicated by systemHealthID. This value MUST match one of the message type values previously supplied by the IMC to the TNCC in the IMC's most recent call to the TNCC's reportMessageTypes() or reportMessageTypesLong() methods. IMCs MAY check these parameters to make sure they are valid and throw an exception if not, but IMCs are not required to make these checks.

The IMC MUST NOT send any IMC-IMV messages on the connection after this method is called, since IF-TNCCS-SOH supports only one round-trip.

As with all IMC methods, the IMC 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 IMC MUST NOT ever modify the buffer contents and MUST NOT access the buffer after receiveMessageSOH has returned. If the IMC wants to retain the contents of the SOHRReportEntry, it should copy it before returning from receiveMessageSOH.

Parameters:
`c` - the IMC connection object
`systemHealthID` - the type of message to be delivered
`sohrReportEntry` - the message to be delivered
Throws:
`TNCException` - if a TNC error occurs

4.3.12.8 IMCLong Interface (IMCLong.java)

org.trustedcomputinggroup.tnc.ifimc

Interface IMCLong

`public interface IMCLong`
This interface can be implemented by an object that already supports the IMC interface to indicate that this object is an IMC 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(IMCConnection c,
                        long messageFlags,
                        long messageVendorID,
                        long messageSubtype,
                        byte[] message,
                        long sourceIMVID,
                        long destinationIMCID)
                        throws TNCException
```

The TNC Client calls this method (if supported by the TNCC and implemented by an IMC) to deliver a message to the IMC. 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 IMV ID and 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. The message must be from an IMV (or a TNCS or other party acting as an IMV). Any flags associated with the message are included in the `messageFlags` parameter. The `sourceIMVID` and `destinationIMCID` parameters indicate the IMV ID of the IMV that sent this message (if available) and either the IMC ID of the intended recipient (if the EXCL flag is set), or the IMC ID in response to whose message or messages this message was sent. If the EXCL flag is set, `destinationIMCID` MUST be either the primary IMC ID for this IMC (matching the value of the `TNC_AttributeID_PRIMARY_IMC_ID` attribute) or an additional IMC ID reserved when the IMC requested the TNCC to do so by calling `reserveAdditionalIMCID`. If the EXCL flag is not set, then `destinationIMCID` MAY be set to the wild card `TNC_IMCID_ANY`.

If an IF-TNCCS protocol that support long types or exclusive delivery is used for a connection, and the IMC and TNCC implement `receiveMessageLong`, the TNCC SHOULD use this method to deliver messages instead of using `receiveMessage`. However, if the IMC does not implement `receiveMessageLong`, the TNCC 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 IMC that does not support `receiveMessageLong`. This should be fine, since the IMC in question wouldn't have the ability to process such messages. Also, the IMC 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 IMV IDs can be handled using the `receiveMessage` method.

The IMC 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 IMC methods, the IMC 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).

IMCs are not required to implement this method, but any IMC that implements the IMCLong interface MUST implement this method. An IMC should implement this method if it wants to receive messages with long types or flags or source IMV IDs. Simple IMCs need not implement this method. Since this method was not included in IF-IMC 1.2, many IMCs do not implement it. TNCCs MUST work properly if an IMC does not implement this method. Likewise, many TNCCs do not implement this method, so IMCs MUST work properly if a TNCC does not implement this method.

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

In the `messageVendorID` and `messageSubtype` parameters, the TNCC MUST pass the vendor ID and message subtype of the message. These values MUST match one of the values previously supplied by the IMC to the TNCC in the IMC's most recent call to `reportMessageTypes` or `reportMessageTypesLong`. The TNCC 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. IMCs MAY check these parameters to make sure they are valid and throw an exception if not, but IMCs 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 TNCC MUST process that flag itself to ensure that a message with this flag set is only delivered to the intended recipient.

Parameters:
`c` - the IMC connection 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

`sourceIMVID` – source IMV ID for the message
`destinationIMCID` - destination IMC ID for message
Throws:
`TNCException` - if a TNC error occurs

4.3.12.9 IMCConnection Interface (IMCConnection.java)

org.trustedcomputinggroup.tnc.ifimc

Interface IMCConnection

`public interface` **`IMCConnection`**

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

The TNCC MUST create a new IMCConnection object for each combination of an IMC and a connection. IMCConnection objects MUST NOT be shared between multiple IMCs.

Field Detail

`static final long` **`TNC_RETRY_REASON_IMC_REMEDIATION_COMPLETE`**
Handshake retry reason when IMC has completed remediation.

`static final long` **`TNC_RETRY_REASON_IMC_SERIOUS_EVENT`**
Handshake retry reason when IMC has detected a serious event and recommends handshake retry even if network connectivity must be interrupted.

`static final long` **`TNC_RETRY_REASON_IMC_INFORMATIONAL_EVENT`**
Handshake retry reason when IMC has detected an event that it would like to communicate to the IMV. It requests handshake retry but not if network connectivity must be interrupted.

`static final long` **`TNC_RETRY_REASON_IMC_PERIODIC`**
Handshake retry reason when IMC wishes to conduct a periodic recheck. It recommends handshake retry but not if network connectivity must be interrupted.

Method Detail

```
void sendMessage(long messageType,
                 byte[] message)
                 throws TNCException
```

Gives a message to the TNCC 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 IMCConnections MUST implement this method. An IMCConnection MUST NOT ever modify the buffer contents and MUST NOT access the buffer after this method has returned. The IMCConnection will typically copy the message out of the buffer, queue it up for delivery, and return from this method.

The IMC MUST NOT call this method unless it has received a call to the IMC's beginHandshake method, the IMC's receiveMessage method, or the IMC's batchEnding method for this connection and the IMC has not yet returned from that method. If the IMC violates this prohibition, this method SHOULD throw a TNCException with result code TNC_RESULT_ILLEGAL_OPERATION. If an IMC really wants to communicate with an IMV at another time, it should call the IMCConnection'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 IMCConnection's sendMessage method will throw a TNCException with result code TNC_RESULT_ILLEGAL_OPERATION. If the TNCC supports limiting the message size or number of round trips, the TNCC 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 IMC can get the Maximum Message Size and Maximum Number of Round Trips by using the related Attribute ID with the getAttribute method on the IMCConnection object. The IMC SHOULD adapt its behavior to accommodate these limitations if available.

The TNC Client MUST support any message type. However, the IMC 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. If the IMC violates this prohibition, the IMCConnection SHOULD throw a TNCException with result code TNC_RESULT_INVALID_PARAMETER. If the IMCConnection supports limiting the message size or number of round trips, the IMCConnection MUST return TNC_RESULT_EXCEEDED_MAX_MESSAGE_SIZE exception or TNC_RESULT_EXCEEDED_MAX_ROUND_TRIPS exception respectively if the limits are exceeded. An IMC can get the Maximum Message Size and Maximum Number of Round Trips by using the related TNC_ATTRIBUTEID within the AttributeSupport.GetAttribute method. The IMC 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 (may be null)
Throws:
`TNCException` - if an error occurs

```
void requestHandshakeRetry(long reason)
                           throws TNCException
```

Asks an IMCConnection to retry an Integrity Check Handshake. The IMC MUST pass one of the handshake retry reasons listed in IF-IMC Abstract API as the reason parameter.

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

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

Parameters:
`reason` - the reason for the handshake request
Throws:
`TNCException` - if an error occurs

4.3.12.10 IMCConnectionLong Interface (IMCConnectionLong.java)

org.trustedcomputinggroup.tnc.ifimc

Interface IMCConnectionLong

`public interface` **`IMCConnectionLong`**

This interface can be implemented by an object that already supports the IMCConnection interface to indicate that this object is an IMCConnection 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 sourceIMCID,
                     long destinationIMVID)
                     throws TNCException
```

An IMC calls this method to give a message to the TNCC 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 IMC ID and a desired or exclusive 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. TNCCs MAY check these values to make sure they are valid and throw an exception if not, but TNCCs 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 IMC is requesting that the message only be delivered to the IMV designated by the `destinationIMVID` parameter. If the bit is cleared, the message is to be delivered to any IMV that has indicated an interest in the message type. Other flags may be defined in future versions of this specification. Until that time, IMCs MUST leave all these flags cleared. Some connections and/or TNCCs may not support the EXCL flag. To determine whether a particular TNCC and connection supports this flag, the IMC should get the Has Exclusive attribute for that connection. If the IMC sets a flag that is not supported by the TNCC or the connection, the TNCC SHOULD throw a `TNCException` with a `TNC_RESULT_INVALID_PARAMETER` result.

The IMC MUST set the `sourceIMCID` parameter to either the primary IMC ID queried by IMC using the `TNC_AttributeID_PRIMARY_IMC_ID` attribute or an additional IMC ID reserved when IMC requests the TNCC to do so by calling `reserveAdditionalIMCID`. This IMC ID will be used as the sourceIMCID for protocols that support sending this field along with the message.

If the EXCL flag is set, the IMC MUST set the `destinationIMVID` parameter to indicate which IMC should receive the message being sent. The IMC should place into this parameter a value which it has obtained using the `sourceIMCID` parameter for messages previously delivered to the IMC on this connection. The IMC MAY set the `destinationIMVID` parameter to a specific IMV 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 IMV indicated in the `destinationIMVID` parameter. If the IMC does not know the IMV ID of the recipient IMV, the IMC MUST use the `TNC_IMVID_ANY` wild card for the `destinationIMVID` parameter and MUST NOT set the EXCL flag. In such case, the message is delivered to all IMVs that have expressed interest in receiving such messages. This may happen when the TNCC starts the integrity handshake and the IMC does not know the IMV IDs of the recipient IMVs.

TNCCs are not required to implement this method, but any TNCC that implements the IMCConnectionLong interface MUST implement this method. Since this method was not included in IF-IMC 1.0 through 1.2, many TNCCs do not implement it. IMCs MUST work properly if a TNCC does not implement this method. The IMC is never required to call this method. The TNCC MUST work with IMCs that don't call this method. The IMC 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 IMC MAY call this method (if implemented) but MUST NOT use a `messageVendorID` greater than `0xffffff` or a `messageSubtype` greater than `0xff`. If the IMC does so, the TNCC SHOULD throw a `TNC_RESULT_NO_LONG_TYPES` exception.

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

The IMC MUST NOT call this method unless it has received a call to `beginHandshake`, `receiveMessage`, `receiveMessageSOH`, `receiveMessageLong`, or `batchEnding` for this connection and the IMC has not yet returned from that method. If the IMC violates this prohibition, the TNCC SHOULD throw a `TNC_RESULT_ILLEGAL_OPERATION` exception. If an IMC really wants to communicate with an IMV 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 TNCC will throw a `TNC_RESULT_ILLEGAL_OPERATION` exception from `sendMessageLong`. If the TNCC supports limiting the message size or number of round trips, the TNCC MUST throw a

`TNC_RESULT_EXCEEDED_MAX_MESSAGE_SIZE` or `TNC_RESULT_EXCEEDED_MAX_ROUND_TRIPS` exception respectively if the limits are exceeded. An IMC can get the Maximum Message Size and Maximum Number of Round Trips by using the related Attribute ID within the `getAttribute` method. The IMC SHOULD adapt its behavior to accommodate these limitations if available.

The TNC Client MUST support any message type. However, the IMC 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 IMC violates this prohibition, the TNCC 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 IMCConnectionSOH Interface (IMCConnectionSOH.java)

org.trustedcomputinggroup.tnc.ifimc

Interface IMCConnectionSOH

`public interface` **`IMCConnectionSOH`**
This interface can be implemented by an object that already supports the IMCConnection interface to indicate that this object is an IMCConnection 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[] sohReportEntry)
                    throws TNCException
```

An IMC calls this method (if implemented by the TNCC) to give a SoHReportEntry to the TNCC for delivery. This allows an IMC to send an entire SoHReportEntry instead of just the contents of the Vendor-Specific attribute, as would be the case if the `sendMessage` method was used.

The SoHReportEntry (as defined in section 3.5 of [9]) is contained in the buffer referenced by the `sohReportEntry` parameter. If the length of the `sohReportEntry` parameter is zero (0), the TNCC MUST throw a `TNCException` with a `TNC_RESULT_INVALID_PARAMETER` result since a zero-length SoHReportEntry is prohibited by the IF_TNCCS-SOH 1.0 specification. No type is included with the SoHReportEntry since this is included in the System-Health-ID attribute in the SoHReportEntry. TNCCs MAY check these values to make sure they are valid and throw a `TNCException` with a `TNC_RESULT_INVALID_PARAMETER` result if not, but TNCCs are not required to make these checks.

A TNCC that supports IF-TNCCS-SOH SHOULD support this method. The TNC Client MUST NOT ever modify the buffer contents and MUST NOT access the buffer after `sendMessageSOH` has returned. The TNC Client will typically copy the SoHReportEntry out of the buffer, place it into the SoH message or queue it up for such placement, and return from this method.

An IMC may call this method if it needs to send an SoHReportEntry with a connection that supports IF-TNCCS-SOH (as indicated by the Has SOH attribute being set to a value of 1). If an IMC calls this method for a connection that does not support IF-TNCCS-SOH, the TNCC SHOULD throw a `TNCException` with a `TNC_RESULT_NO_SOH_SUPPORT` result. IMCs should recognize that many TNCCs do not support IF-TNCCS-SOH and some TNCCs that do support IF-TNCCS-SOH will not support this method. Therefore, IMCs should be prepared to send messages via `sendMessage` in some cases when IF-TNCC-SOH is used. Simple IMCs 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 SoHReportEntry.

The IMC MUST NOT call this method unless it has received a call to `beginHandshake`, `receiveMessage`, `receiveMessageSOH`, `receiveMessageLong`, or `batchEnding` for this connection and the IMC has not yet returned from that method. If the IMC violates this prohibition, the TNCC SHOULD throw a `TNC_RESULT_ILLEGAL_OPERATION` exception. If an IMC really wants to communicate with an IMV 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 TNCC will throw a `TNCException` with a `TNC_RESULT_ILLEGAL_OPERATION` result.

If the TNCC supports limiting the message size or number of round trips and the limits are exceeded, the TNCC MUST throw a `TNCException` with a result of `TNC_RESULT_EXCEEDED_MAX_MESSAGE_SIZE` or `TNC_RESULT_EXCEEDED_MAX_ROUND_TRIPS`, as the case may be. An IMC can get the Maximum Message Size and Maximum Number of Round Trips by using the related Attribute ID within the getAttribute method. The IMC SHOULD adapt its behavior to accommodate these limitations if available.

Parameters:
`sohReportEntry` - sohReportEntry to be delivered
Throws:
`TNCException` - if a TNC error occurs

5 Security Considerations

IF-IMC is a critical component since it enables third party security applications to provide the information that can be used by administrators to make network access control decisions. The security of this interface is critical to ensure that the TNC framework is not itself subject to attack and circumvention. Hostile code has many ways to enter a platform and can eliminate, tamper with or circumvent security applications.

This section describes the security threats related to IF-IMC and suggests methods to address these threats. The components involved in IF-IMC are one or more Trusted Network Connect Clients (TNCC) and one or more Integrity Measurement Collectors (IMCs). All the above components reside on the same endpoint or Access Requestor (AR). IF-IMC is the interface between the TNCC and the IMCs.

5.1 Threat Analysis

5.1.1 Registration and Discovery Based Threats

The TNCC discovers which IMCs are installed on a platform via a platform specific binding, for example, on the Windows platform using a windows registry key and on the Linux or Unix platform, a configuration file. On Windows, the registry keys are typically created when the IMCs are installed, requiring the IMC installer to possess sufficient privileges on the platform. Similarly the TNCC must have sufficient privileges to read the relevant keys. Based on the IMCs discovered in the registry, the TNCC 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 IMC
- It can remove a valid IMC, perhaps replacing it with rogue/modified versions of code

Similar attacks can also be mounted by modifying the code of an IMC or critical data upon which the IMC depends.

The ability to add an invalid IMC can have considerable impact, as detailed in the next section.

5.1.2 Rogue IMC Threats

If a rogue IMC is installed and then loaded by a valid TNCC, it may be able to misuse the IF-IMC API in the following ways:

- Overwrite TNCC or IMC memory
- Violate IF-IMC API requirements such as passing illegal or unexpected argument values
- Perform illegal operations so that the TNCC is terminated by the operating system
- Perform improper operations with the TNCC's privileges
- Attack other components (such as the NAR or applications) using the privileges or credentials of the TNCC or other IMCs
- 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 this or other ARs
- Issue a large number of interface API calls to the TNCC (Denial of service of the TNCC)
- Spoof specific IMCs and provide incorrect information to a TNCC about other IMCs
- Spoof TNCC calls to a IMC and provide incorrect connection notification changes.
- Spoof specific IF-IMC APIs and provide incorrect received messages or request incorrect message types for other IMCs
- Spuriously request handshake retries (Denial of service)
- Lock up TNCC threads by not returning from function calls (Denial of service)
- Cause untimely unloading of IMCs
- Use vendor-specific extensions to IF-IMV to perform other attacks

5.1.3 Rogue TNCC Threats

If a rogue TNCC loads a valid IMC, it may be able to misuse IF-IMC in the following ways:

- Overwrite IMC memory
- Violate IF-IMC API requirements such as passing illegal or unexpected argument values
- Attack other components (such as the NAR or applications) using the credentials of an IMC
- 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 this or other ARs
- Issue a large number of, or particularly expensive, interface API calls to an IMC, possibly causing denial of service of a critical security application
- Spuriously request or perform handshake retries (Denial of service)
- Use vendor-specific extensions to IF-IMC to perform other attacks

5.1.4 Man-in-the-Middle Threats

If an attacker injects a man-in-the-middle between an IMC and its corresponding IMV peer entity, between an IMC and its corresponding TNCC, or between a TNCC and a TNCS it may be able to be misused in the following ways:

- Allows the viewing, modification, deletion, or addition, of messages passing between the IMC and the IMV, between the IMC and its corresponding TNCC, or between the TNCC and the TNCS,
- 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 IMCs and TNCCs

Malicious code (worms, viruses, etc) or another unauthorized application can modify an IMC or TNCC. 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-IMC

IF-IMC is part of the larger TNC architecture. Successful attacks against other parts of the TNC architecture will generally result in negative effects for IMCs, TNCCs, 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 IMCs be loaded by a TNCC and only authorized TNCCs be allowed to load an IMC. There are well known methods to control what code is loaded by a TNCC:

- 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 IMC registration mechanism (registry or configuration file)
- Control access to IMC code and critical data files
- Employ a TNCC-specific list of authorized IMCs

Similar checks can be performed by the operating system before loading the TNCC.

Industry standard best practices for secure coding, software engineering, and code reviews should be followed during the development of IMCs and TNCCs 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 IMCs can be monitored.

However, stronger protection against rogue IMCs can be provided by having the TNCC launch a new "child" process for each IMC, having the child process load the IMC, and then having the TNCC communicate with the child processes carefully. This limits the amount of damage that can be done by a rogue IMC. The TNCC may use this approach but is not required to do so.

This specification requires that all valid IMCs be installed to a protected system directory. The loading of a rogue IMC 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 IMC to be installed.

IMC implementers who choose a stub-to-application implementation must take care not to make the stub-to-application 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 IMC and the IMV) and/or the IF-TNCCS (between the TNCC and the TNCS 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 IMCs and TNCCs 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-IMC 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 IMCs and TNC Clients on their platform.

```
/* tncifimc.h
 *
 * Trusted Network Connect IF-IMC API version 1.30
 * Microsoft Windows DLL Platform Binding C Header
 * February 25, 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 _TNCIFIMC_H
#define _TNCIFIMC_H

#ifdef __cplusplus
extern "C" {
```

```
#endif

#ifdef WIN32
#ifdef TNC_IMC_EXPORTS
#define TNC_IMC_API __declspec(dllexport)
#else
#define TNC_IMC_API __declspec(dllimport)
#endif
#else
#define TNC_IMC_API
#endif

/* Basic Types */

typedef unsigned long TNC_UInt32;
typedef unsigned char *TNC_BufferReference;

/* Derived Types */

typedef TNC_UInt32 TNC_IMCID;
typedef TNC_UInt32 TNC_ConnectionID;
typedef TNC_UInt32 TNC_ConnectionState;
typedef TNC_UInt32 TNC_RetryReason;
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_IMC_InitializePointer)(
    TNC_IMCID imcID,
    TNC_Version minVersion,
    TNC_Version maxVersion,
    TNC_Version *pOutActualVersion);
typedef TNC_Result (*TNC_IMC_NotifyConnectionChangePointer)(
    TNC_IMCID imcID,
    TNC_ConnectionID connectionID,
    TNC_ConnectionState newState);
typedef TNC_Result (*TNC_IMC_BeginHandshakePointer)(
    TNC_IMCID imcID,
    TNC_ConnectionID connectionID);
typedef TNC_Result (*TNC_IMC_ReceiveMessagePointer)(
    TNC_IMCID imcID,
    TNC_ConnectionID connectionID,
    TNC_BufferReference message,
    TNC_UInt32 messageLength,
    TNC_MessageType messageType);
typedef TNC_Result (*TNC_IMC_ReceiveMessageSOHPointer)(
    TNC_IMCID imcID,
    TNC_ConnectionID connectionID,
    TNC_BufferReference sohrReportEntry,
    TNC_UInt32 sohrRELength,
    TNC_MessageType systemHealthID);
typedef TNC_Result (*TNC_IMC_ReceiveMessageLongPointer)(
    TNC_IMCID imcID,
    TNC_ConnectionID connectionID,
    TNC_UInt32 messageFlags,
    TNC_BufferReference message,
    TNC_UInt32 messageLength,
    TNC_VendorID messageVendorID,
    TNC_MessageSubtype messageSubtype,
    TNC_UInt32 sourceIMVID,
    TNC_UInt32 destinationIMCID);
typedef TNC_Result (*TNC_IMC_BatchEndingPointer)(
    TNC_IMCID imcID,
    TNC_ConnectionID connectionID);
typedef TNC_Result (*TNC_IMC_TerminatePointer)(
    TNC_IMCID imcID);
typedef TNC_Result (*TNC_TNCC_ReportMessageTypesPointer)(
    TNC_IMCID imcID,
    TNC_MessageTypeList supportedTypes,
    TNC_UInt32 typeCount);
typedef TNC_Result (*TNC_TNCC_ReportMessageTypesLongPointer)(
    TNC_IMCID imcID,
    TNC_VendorIDList supportedVendorIDs,
    TNC_MessageSubtypeList supportedSubtypes,
    TNC_UInt32 typeCount);
typedef TNC_Result (*TNC_TNCC_SendMessagePointer)(
    TNC_IMCID imcID,
    TNC_ConnectionID connectionID,
    TNC_BufferReference message,
    TNC_UInt32 messageLength,
    TNC_MessageType messageType);
typedef TNC_Result (*TNC_TNCC_SendMessageSOHPointer)(
    TNC_IMCID imcID,
    TNC_ConnectionID connectionID,
    TNC_BufferReference sohReportEntry,
    TNC_UInt32 sohRELength);
typedef TNC_Result (*TNC_TNCC_SendMessageLongPointer)(
    TNC_IMCID imcID,
    TNC_ConnectionID connectionID,
    TNC_UInt32 messageFlags,
    TNC_BufferReference message,
    TNC_UInt32 messageLength,
    TNC_VendorID messageVendorID,
    TNC_MessageSubtype messageSubtype,
    TNC_UInt32 destinationIMVID);
typedef TNC_Result (*TNC_TNCC_RequestHandshakeRetryPointer)(
    TNC_IMCID imcID,
    TNC_ConnectionID connectionID,
    TNC_RetryReason reason);
typedef TNC_Result (*TNC_TNCC_GetAttributePointer)(
    TNC_IMCID imcID,
    TNC_ConnectionID connectionID,
    TNC_AttributeID attributeID,
    TNC_UInt32 bufferLength,
    TNC_BufferReference buffer,
    TNC_UInt32 *pOutValueLength);
```

```
typedef TNC_Result (*TNC_TNCC_SetAttributePointer)(
    TNC_IMCID imcID,
    TNC_ConnectionID connectionID,
    TNC_AttributeID attributeID,
    TNC_UInt32 bufferLength,
    TNC_BufferReference buffer);
typedef TNC_Result (*TNC_TNCC_ReserveAdditionalIMCIDPointer)(
    TNC_IMCID imcID,
    TNC_UInt32 *pOutIMCID);
typedef TNC_Result (*TNC_TNCC_BindFunctionPointer)(
    TNC_IMCID imcID,
    char *functionName,
    void **pOutfunctionPointer);
typedef TNC_Result (*TNC_IMC_ProvideBindFunctionPointer)(
    TNC_IMCID imcID,
    TNC_TNCC_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
#define 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_IFIMC_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 */

#define TNC_RETRY_REASON_IMC_REMEDIATION_COMPLETE 0
#define TNC_RETRY_REASON_IMC_SERIOUS_EVENT 1
```

```
#define TNC_RETRY_REASON_IMC_INFORMATIONAL_EVENT 2
#define TNC_RETRY_REASON_IMC_PERIODIC 3
/* reserved for TNC_RETRY_REASON_IMV_IMPORTANT_POLICY_CHANGE: 4 */
/* reserved for TNC_RETRY_REASON_IMV_MINOR_POLICY_CHANGE: 5 */
/* reserved for TNC_RETRY_REASON_IMV_SERIOUS_EVENT: 6 */
/* reserved for TNC_RETRY_REASON_IMV_MINOR_EVENT: 7 */
/* reserved for TNC_RETRY_REASON_IMV_PERIODIC: 8 */

/* 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)
/* reserved for TNC_ATTRIBUTEID_REASON_STRING (0x00000002) */
/* reserved for TNC_ATTRIBUTEID_REASON_LANGUAGE (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_SOHR ((TNC_AttributeID) 0x00559708)
#define TNC_ATTRIBUTEID_SSOHR ((TNC_AttributeID) 0x00559709)
#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_IMC_SPTS_TNCS1 ((TNC_AttributeID) 0x0055970F)
#define TNC_ATTRIBUTEID_PRIMARY_IMC_ID ((TNC_AttributeID) 0x00559711)

/* IMC Functions */

TNC_IMC_API TNC_Result TNC_IMC_Initialize(
/*in*/  TNC_IMCID imcID,
/*in*/  TNC_Version minVersion,
/*in*/  TNC_Version maxVersion,
/*out*/ TNC_Version *pOutActualVersion);
```

```
TNC_IMC_API TNC_Result TNC_IMC_NotifyConnectionChange(
/*in*/  TNC_IMCID imcID,
/*in*/  TNC_ConnectionID connectionID,
/*in*/  TNC_ConnectionState newState);

TNC_IMC_API TNC_Result TNC_IMC_BeginHandshake(
/*in*/  TNC_IMCID imcID,
/*in*/  TNC_ConnectionID connectionID);

TNC_IMC_API TNC_Result TNC_IMC_ReceiveMessage(
/*in*/  TNC_IMCID imcID,
/*in*/  TNC_ConnectionID connectionID,
/*in*/  TNC_BufferReference messageBuffer,
/*in*/  TNC_UInt32 messageLength,
/*in*/  TNC_MessageType messageType);

TNC_IMC_API TNC_Result TNC_IMC_ReceiveMessageSOH(
/*in*/  TNC_IMCID imcID,
/*in*/  TNC_ConnectionID connectionID,
/*in*/  TNC_BufferReference sohrReportEntry,
/*in*/  TNC_UInt32 sohrRELength,
/*in*/  TNC_MessageType systemHealthID);

TNC_IMC_API TNC_Result TNC_IMC_ReceiveMessageLong(
/*in*/  TNC_IMCID imcID,
/*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 sourceIMVID,
/*in*/  TNC_UInt32 destinationIMCID);

TNC_IMC_API TNC_Result TNC_IMC_BatchEnding(
/*in*/  TNC_IMCID imcID,
/*in*/  TNC_ConnectionID connectionID);

TNC_IMC_API TNC_Result TNC_IMC_Terminate(
/*in*/  TNC_IMCID imcID);

TNC_IMC_API TNC_Result TNC_IMC_ProvideBindFunction(
/*in*/  TNC_IMCID imcID,
/*in*/  TNC_TNCC_BindFunctionPointer bindFunction);

/* TNC Client Functions */

TNC_Result TNC_TNCC_ReportMessageTypes(
/*in*/  TNC_IMCID imcID,
/*in*/  TNC_MessageTypeList supportedTypes,
/*in*/  TNC_UInt32 typeCount);

TNC_Result TNC_TNCC_ReportMessageTypesLong(
/*in*/  TNC_IMCID imcID,
/*in*/  TNC_VendorIDList supportedVendorIDs,
/*in*/  TNC_MessageSubtypeList supportedSubtypes,
/*in*/  TNC_UInt32 typeCount);

TNC_Result TNC_TNCC_SendMessage(
/*in*/  TNC_IMCID imcID,
/*in*/  TNC_ConnectionID connectionID,
/*in*/  TNC_BufferReference message,
/*in*/  TNC_UInt32 messageLength,
/*in*/  TNC_MessageType messageType);

TNC_Result TNC_TNCC_SendMessageSOH(
/*in*/  TNC_IMCID imcID,
/*in*/  TNC_ConnectionID connectionID,
/*in*/  TNC_BufferReference sohReportEntry,
/*in*/  TNC_UInt32 sohRELength);

TNC_Result TNC_TNCC_SendMessageLong(
/*in*/  TNC_IMCID imcID,
/*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 destinationIMVID);
TNC_Result TNC_TNCC_RequestHandshakeRetry(

/*in*/  TNC_IMCID imcID,
/*in*/  TNC_ConnectionID connectionID,
/*in*/  TNC_RetryReason reason);

TNC_Result TNC_TNCC_GetAttribute(
/*in*/  TNC_IMCID imcID,
/*in*/  TNC_ConnectionID connectionID,
/*in*/  TNC_AttributeID attributeID,
/*in*/  TNC_UInt32 bufferLength,
/*out*/ TNC_BufferReference buffer,
/*out*/ TNC_UInt32 *pOutValueLength);


TNC_Result TNC_TNCC_SetAttribute(
/*in*/  TNC_IMCID imcID,
/*in*/  TNC_ConnectionID connectionID,
/*in*/  TNC_AttributeID attributeID,
/*in*/  TNC_UInt32 bufferLength,
/*in*/  TNC_BufferReference buffer);

TNC_Result TNC_TNCS_ReserveAdditionalIMCID(
/*in*/  TNC_IMCID imcID,
/*out*/ TNC_UInt32 *pOutIMCID);

TNC_Result TNC_TNCC_BindFunction(
/*in*/  TNC_IMCID imcID,
/*in*/  char *functionName,
/*out*/ 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-IMC supports the use case. The text describing IF-IMC 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

1. 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 client-side components (TNCC and IMCs) find each other automatically using Microsoft Windows registry or a configuration file modified at install time. The manner in which the NAR and TNCC find each other is not specified.
2. 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.

   The TNCS initializes the IMVs through IF-IMV.

7.3 TNCC Startup

1. When the TNCC starts up, the TNCC loads the IMCs. **[IF-IMC] The details of the load process are platform-specific. With the Microsoft Windows DLL binding, the TNCC reads a protected registry key to find the IMC DLLs, then loads them.**

   During the load process, the TNCC may check the integrity of the IMCs. This is optional. If a TPM is present, this check will typically involve hashing the IMCs and adding their hashes to a PCR. If no TPM is present, this check may involve checking the signatures on the IMCs. Integrity checks during IMC loading are done completely by the TNCC since there is no TNCS or IMV available. TNCS and IMVs will get a chance to do platform authentication of the endpoint platform later on.

2. The TNCC initializes the IMCs through IF-IMC. **[IF-IMC] The TNCC calls `TNC_IMC_Initialize` for each IMC. The IMC performs any initialization it may need to, such as connecting to a background process or starting threads (if permitted by the platform binding).**

3. **[IF-IMC] The TNCC performs any other platform-specific initialization needed. With the Microsoft Windows DLL binding, the TNCC calls the `TNC_IMC_ProvideBindFunction` function to give each IMC a pointer to the bind function (`TNC_TNCC_BindFunction`) used for Dynamic Function Binding.**

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 NAA. The ordering of user authentication, platform authentication, and integrity check is 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 16. 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. **[IF-IMC] If this is a new network connection, the TNCC calls `TNC_IMC_NotifyConnectionChange` with the `newState` parameter set to `TNC_CONNECTION_STATE_CREATE` to indicate that a new network connection has been created. The TNCC calls `TNC_IMC_NotifyConnectionChange` with the `newState` parameter set to `TNC_CONNECTION_STATE_HANDSHAKE` to indicate that a new Integrity Check Handshake is starting. The TNCC calls `TNC_IMC_BeginHandshake` to inform the IMCs that a new Integrity Check Handshake is starting and they should send their messages. The IMCs call `TNC_TNCC_SendMessage` to give their messages to the TNCC and then return from `TNC_IMC_BeginHandshake` to indicate that they are done sending messages for this batch.**
7. The TNCS uses IF-IMV to inform each IMV that an Integrity Check Handshake has started.
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.
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 through IF-IMV. 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.
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. **[IF-IMC] The TNCC delivers the IMV messages to the IMCs via `TNC_IMC_ReceiveMessage`. The IMCs may call `TNC_TNCC_SendMessage` before returning from `TNC_IMC_ReceiveMessage` if they want to send a response. When the TNCC has delivered all the IMV messages to the IMCs, it calls `TNC_IMC_BatchEnding` to inform them of this fact. The IMCs may call `TNC_TNCC_SendMessage` before returning from `TNC_IMC_BatchEnding` if they want to send a message to an IMV.**
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.
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.
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. **[IF-IMC] The TNCC signals this change in network connection state through the `TNC_IMC_NotifyConnectionChange` function.**
20. If the IMCs or the applications that they represent 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. **[IF-IMC] The IMC signals this by calling the `TNC_TNCC_RequestHandshakeRetry` function.**
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 Integrity Check Handshake 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 Integrity Check 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.
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 IMVs are polled for a first batch of messages, which are sent to the TNCC over IF-T and IF-TNCCS.
2. The TNCC delivers the first batch of IMV messages to IMCs that support having the TNCS go first. **[IF-IMC] For all IMCs, the TNCC calls `TNC_IMC_NotifyConnectionChange` with the `newState` parameter set to `TNC_CONNECTION_STATE_HANDSHAKE` to indicate that**

a new Integrity Check Handshake is starting. For IMCs that support having the TNCS go first, the TNCC delivers the IMV messages to the IMCs via `TNC_IMC_ReceiveMessage`. The IMCs may call `TNC_TNCC_SendMessage` before returning from `TNC_IMC_ReceiveMessage` if they want to send a response. When the TNCC has delivered all the IMV messages to the IMCs, it calls `TNC_IMC_BatchEnding` to inform them of this fact. The IMCs may call `TNC_TNCC_SendMessage` before returning from `TNC_IMC_BatchEnding` if they want to send a message to an IMV. For IMCs that do not support having the TNCS go first, the TNCC does not deliver the IMV messages. Instead, it calls `TNC_IMC_BeginHandshake` to inform the IMCs that a new Integrity Check Handshake is starting and they should send their messages.

3. 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 – C Binding: IF-IMC Network Connect Sequence Diagram) illustrates the Network Connect use case, as described in section 7.4.

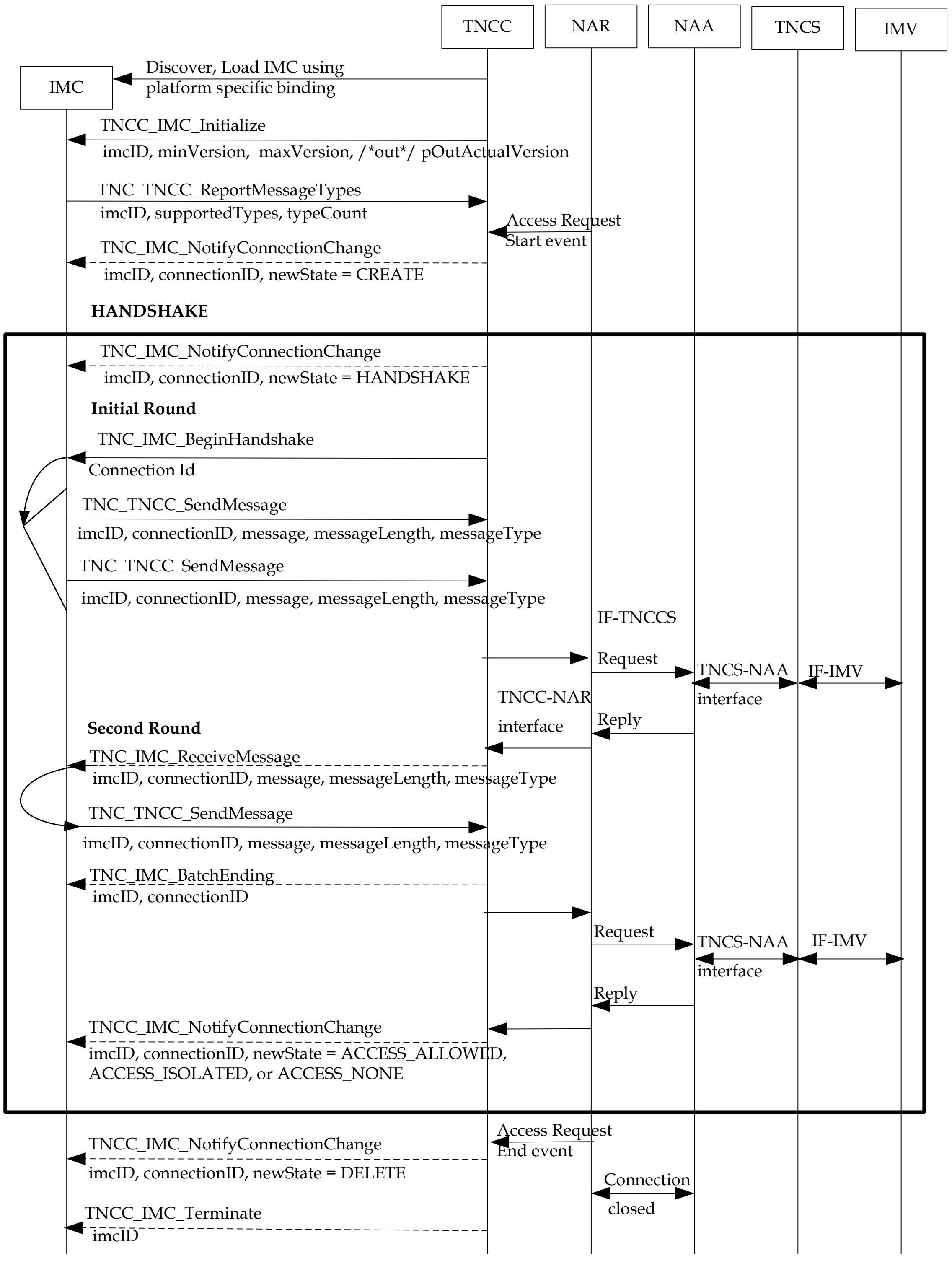

TNCC
NAR
NAA
TNCS
IMV
IMC
Discover, Load IMC using platform specific binding
TNCC_IMC_Initialize
imcID, minVersion, maxVersion, /*out*/ pOutActualVersion
TNC_TNCC_ReportMessageTypes
imcID, supportedTypes, typeCount
Access Request Start event
TNC_IMC_NotifyConnectionChange
imcID, connectionID, newState = CREATE
HANDSHAKE
TNC_IMC_NotifyConnectionChange
imcID, connectionID, newState = HANDSHAKE
Initial Round
TNC_IMC_BeginHandshake
Connection Id
TNC_TNCC_SendMessage
imcID, connectionID, message, messageLength, messageType
TNC_TNCC_SendMessage
imcID, connectionID, message, messageLength, messageType
IF-TNCCS
Request
TNCS-NAA interface
IF-IMV
TNCC-NAR interface
Reply
Second Round
TNC_IMC_ReceiveMessage
imcID, connectionID, message, messageLength, messageType
TNC_TNCC_SendMessage
imcID, connectionID, message, messageLength, messageType
TNC_IMC_BatchEnding
imcID, connectionID
Request
TNCS-NAA interface
IF-IMV
Reply
TNCC_IMC_NotifyConnectionChange
imcID, connectionID, newState = ACCESS_ALLOWED, ACCESS_ISOLATED, or ACCESS_NONE
Access Request End event
TNCC_IMC_NotifyConnectionChange
imcID, connectionID, newState = DELETE
Connection closed
TNCC_IMC_Terminate
imcID

Figure 2 – C Binding: IF-IMC Network Connect Sequence Diagram

7.8.2 Sequence Diagram for Handshake Retry After Remediation

The following sequence diagram (Figure 3 – C Binding: IF-IMC Handshake Retry After Remediation Sequence Diagram) illustrates the Handshake Retry After Remediation use case, as described in section 7.5.

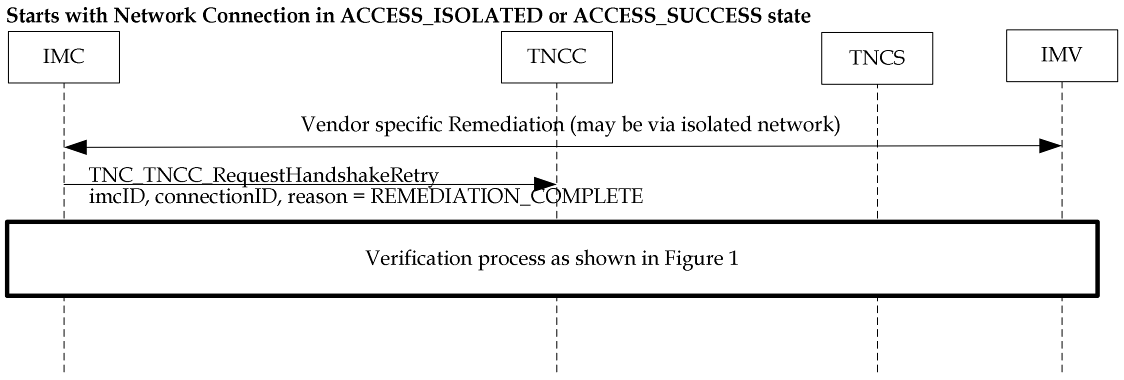

Starts with Network Connection in ACCESS_ISOLATED or ACCESS_SUCCESS state
IMC
TNCC
TNCS
IMV
Vendor specific Remediation (may be via isolated network)
TNC_TNCC_RequestHandshakeRetry
imcID, connectionID, reason = REMEDIATION_COMPLETE
Verification process as shown in Figure 1

Figure 3 – C Binding: IF-IMC 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.

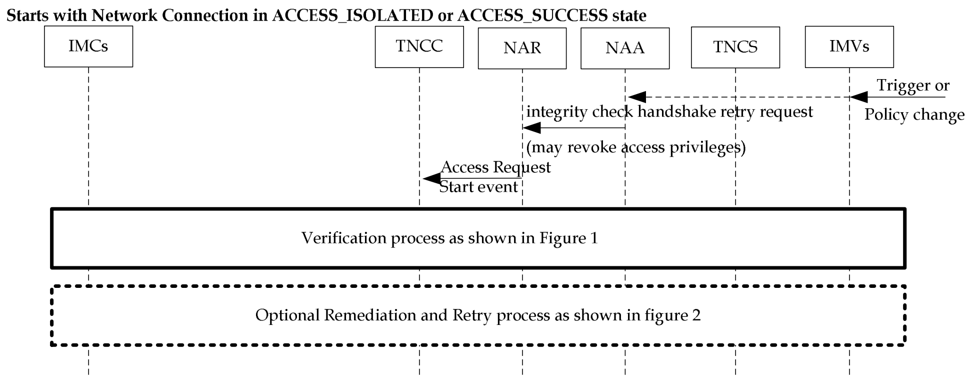

Starts with Network Connection in ACCESS_ISOLATED or ACCESS_SUCCESS state
IMCs
TNCC
NAR
NAA
TNCS
IMVs
Trigger or
Policy change
integrity check handshake retry request
(may revoke access privileges)
Access Request
Start event
Verification process as shown in Figure 1
Optional Remediation and Retry process as shown in figure 2

Figure 4 – C Binding: IF-IMC Handshake Retry Initiated by TNCS Sequence Diagram

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.

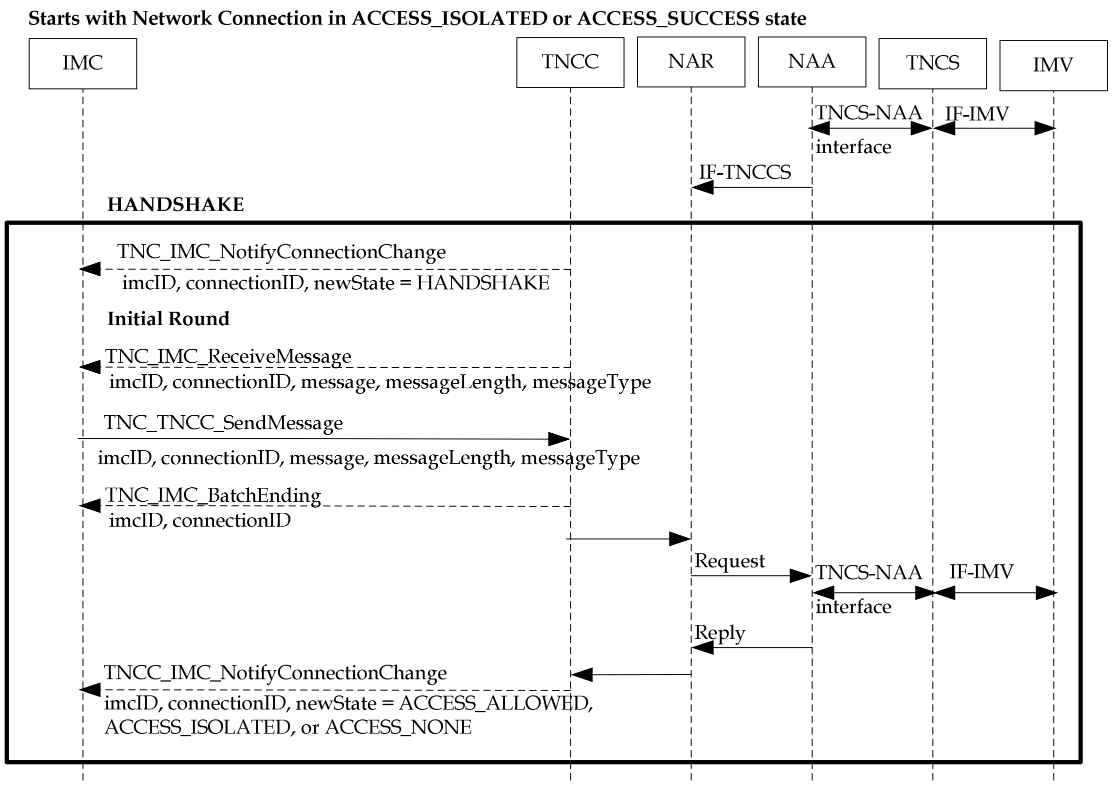

Starts with Network Connection in ACCESS_ISOLATED or ACCESS_SUCCESS state
IMC
TNCC
NAR
NAA
TNCS
IMV
TNCS-NAA
interface
IF-IMV
IF-TNCCS
HANDSHAKE
TNC_IMC_NotifyConnectionChange
imcID, connectionID, newState = HANDSHAKE
Initial Round
TNC_IMC_ReceiveMessage
imcID, connectionID, message, messageLength, messageType
TNC_TNCC_SendMessage
imcID, connectionID, message, messageLength, messageType
TNC_IMC_BatchEnding
imcID, connectionID
Request
TNCS-NAA
interface
IF-IMV
Reply
TNCC_IMC_NotifyConnectionChange
imcID, connectionID, newState = ACCESS_ALLOWED, ACCESS_ISOLATED, or ACCESS_NONE

Figure 5 – C Binding: IF-IMC 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.

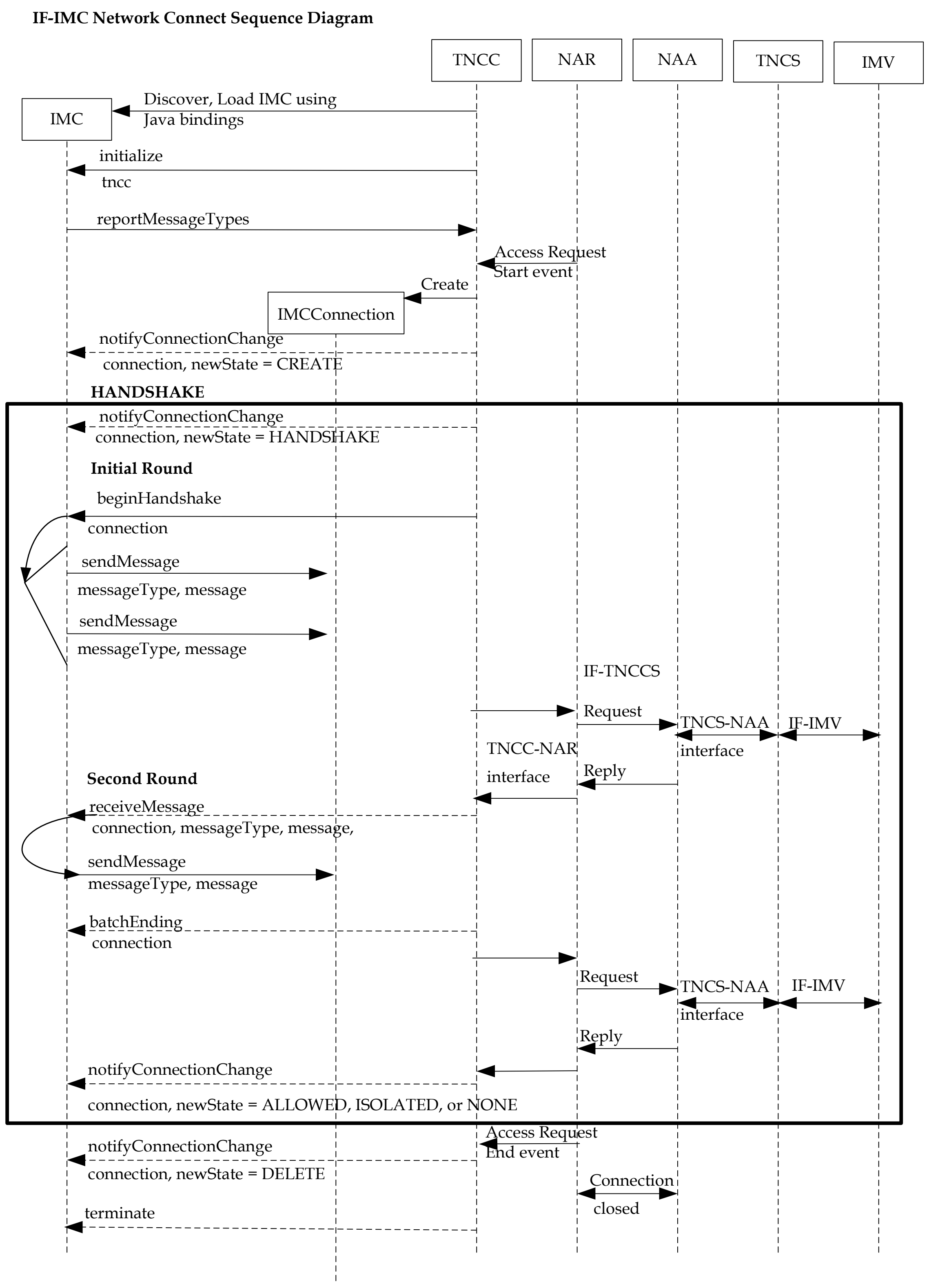


Figure 6 – Java Binding: IF-IMC Network Connect Sequence Diagram

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.

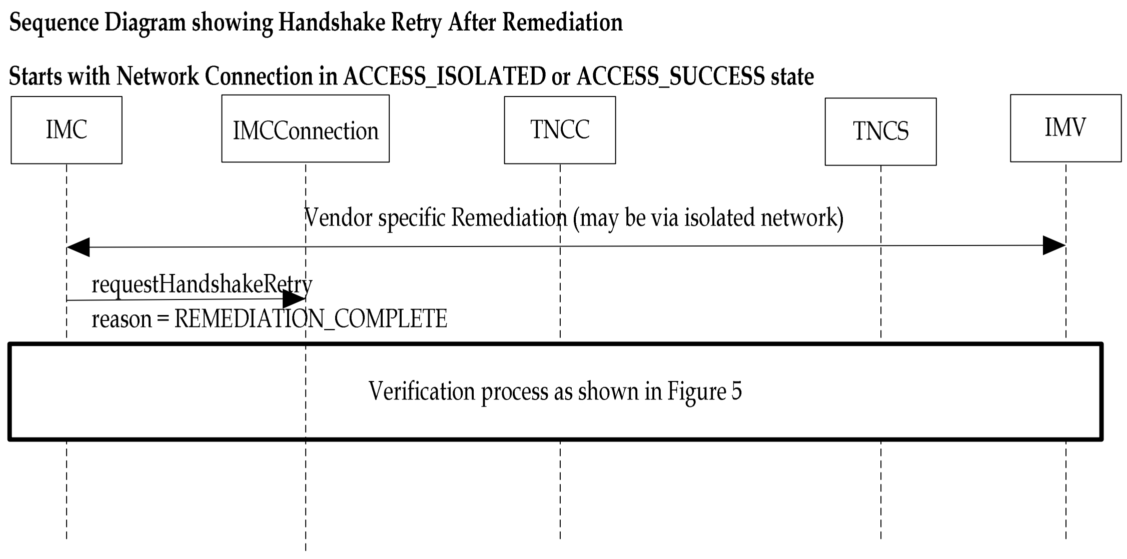

Sequence Diagram showing Handshake Retry After Remediation
Starts with Network Connection in ACCESS_ISOLATED or ACCESS_SUCCESS state
IMC
IMCConnection
TNCC
TNCS
IMV
Vendor specific Remediation (may be via isolated network)
requestHandshakeRetry
reason = REMEDIATION_COMPLETE
Verification process as shown in Figure 5

Figure 7 – Java Binding: IF-IMC 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.

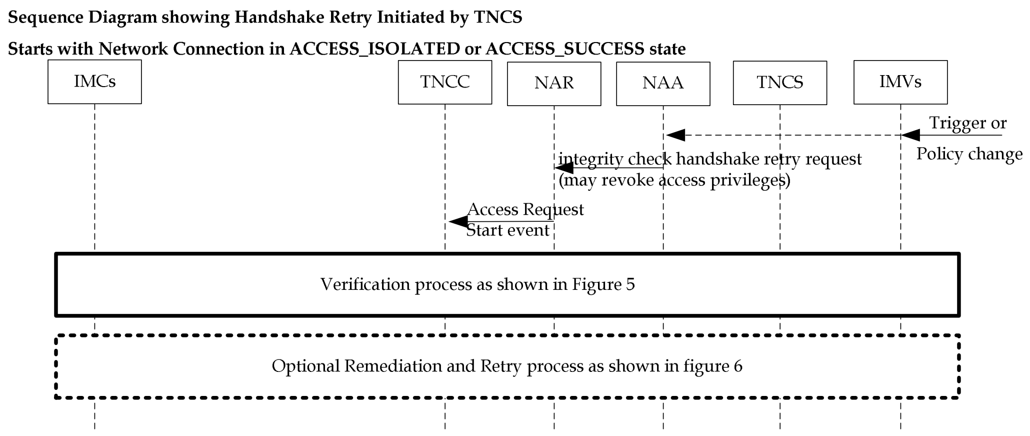

Sequence Diagram showing Handshake Retry Initiated by TNCS
Starts with Network Connection in ACCESS_ISOLATED or ACCESS_SUCCESS state
IMCs
TNCC
NAR
NAA
TNCS
IMVs
Trigger or
Policy change
integrity check handshake retry request
(may revoke access privileges)
Access Request
Start event
Verification process as shown in Figure 5
Optional Remediation and Retry process as shown in figure 6

Figure 8 – Java Binding: IF-IMC Handshake Retry Initiated by TNCS Sequence Diagram

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.

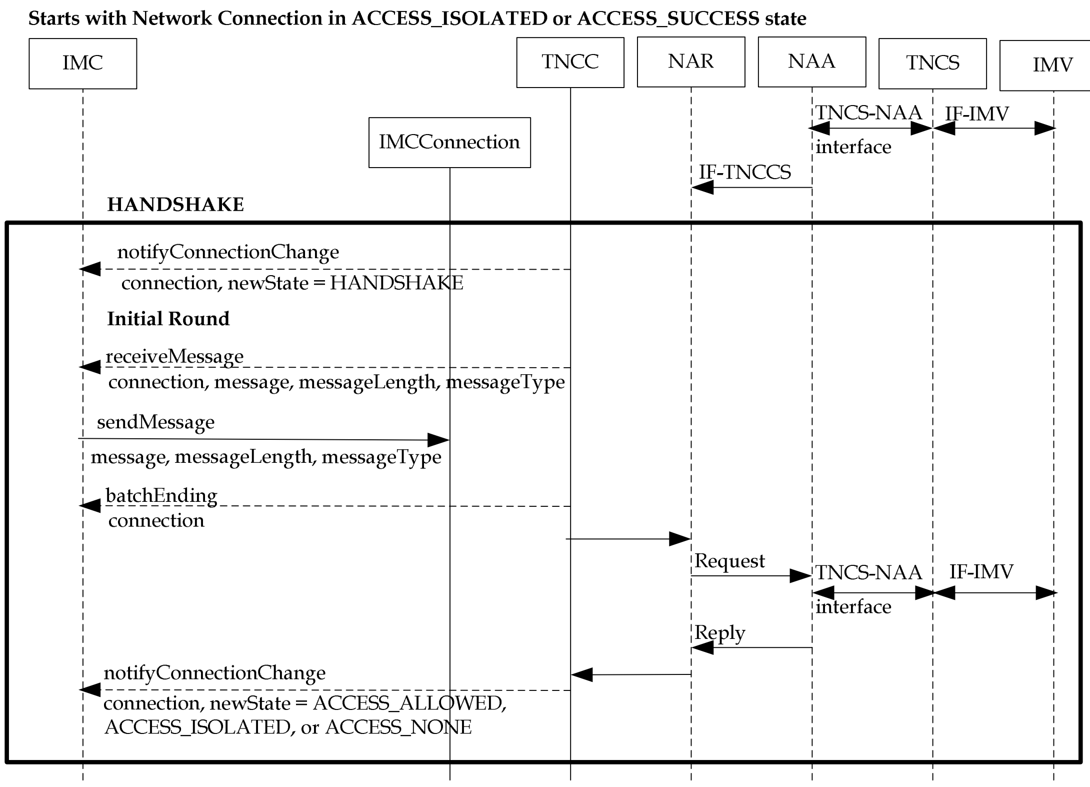

Starts with Network Connection in ACCESS_ISOLATED or ACCESS_SUCCESS state
IMC
TNCC
NAR
NAA
TNCS
IMV
IMCConnection
TNCS-NAA
interface
IF-IMV
IF-TNCCS
HANDSHAKE
notifyConnectionChange
connection, newState = HANDSHAKE
Initial Round
receiveMessage
connection, message, messageLength, messageType
sendMessage
message, messageLength, messageType
batchEnding
connection
Request
TNCS-NAA
interface
IF-IMV
Reply
notifyConnectionChange
connection, newState = ACCESS_ALLOWED,
ACCESS_ISOLATED, or ACCESS_NONE

Figure 9 – Java Binding: IF-IMC Handshake Retry With TNCS First Sequence Diagram

8 Implementing a Simple IMC

This section provides a brief informative (non-binding) description of how to implement a simple IMC, one that only reports a value to an IMV (the operating system version, for instance).

This example assumes that you're using the Microsoft Windows DLL platform binding. If not, replace the instructions in section 8.3 about `TNC_IMC_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 send your value to the IMV 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_IMC_Initialize

All IMCs must implement the `TNC_IMC_Initialize` function. In your implementation, determine whether you support any of the listed IF-IMC API versions. If not, return `TNC_RESULT_NO_COMMON_VERSION`. If so, store the mutually agreed upon version number at `pOutActualVersion` and initialize the IMC. Return `TNC_RESULT_SUCCESS` if all goes well. Normally, you might store your IMC ID for later use but in this example all of your code is called by the TNCC so you have the IMC ID as a parameter to all your functions.

8.3 TNC_IMC_ProvideBindFunction

Use the bind function to get a pointer to `TNC_TNCC_SendMessage` for later use. This is the only state you need to keep. Return `TNC_RESULT_SUCCESS` unless the bind function reports an error. In that case, return `TNC_RESULT_FATAL`.

8.4 TNC_IMC_BeginHandshake

When a new Integrity Check Handshake starts, you just want to send your value and then you're done for the rest of the handshake. To implement this function, call the pointer to `TNC_TNCC_SendMessage` that you saved earlier. Pass in the IMC ID and network connection ID provided to `TNC_IMC_BeginHandshake`, a pointer and length for your message, and the message type you decided on. If `TNC_TNCC_SendMessage` returns an error, then return that. Otherwise, return `TNC_RESULT_SUCCESS`.

8.5 All Done!

That's it! You've implemented your first IMC. If you need to do anything special on termination, you can implement `TNC_IMC_Terminate`. But many IMCs 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] Crocker, D., P. Overell, “Augmented BNF for Syntax Specifications: ABNF”, Internet Engineering Task Force RFC 2234, November 1997.

[4] Alvestrand, H., “Content Language Headers”, Internet Engineering Task Force RFC 3282, May 2002.

9.2 Informative References

[5] Trusted Computing Group, *TNC IF-IMV*, Specification Version 1.3, February 2013.

[6] ISO, ISO/IEC 9899:1999, Programming Languages – C, 1999.

[7] Trusted Computing Group, *TNC IF-T: Protocol Bindings for Tunneled EAP Methods,* Specification v1.0, May 2006.

[8] Trusted Computing Group, *TNC IF-T: Protocol Bindings for Tunneled EAP Methods,* Specification v1.1, May 2007.

[9] Trusted Computing Group, *TNC IF-TNCCS-SOH*, Specification Version 1.0, May 2007.

[10] Trusted Computing Group, *TNC IF-TNCCS: TLV Binding*, Specification Version 2.0, January 2010.

[11] Trusted Computing Group, *TNC IF-TNCCS*, Specification Version 1.0, May 2006.

[12] Trusted Computing Group, *TNC IF-TNCCS*, Specification Version 1.1, February 2007.

[13] Microsoft Corporation, *[MS-PEAP]: Protected Extensible Authentication Protocol (PEAP) Specification*, Version 3.1, March 14, 2008.

[14] Trusted Computing Group, *TCG Attestation PTS Protocol: Binding to TNC IF-M,* Specification v1.0, September 2011.

[15] Trusted Computing Group, *TNC IF-T: Binding to TLS,* Specification Version 1.0, May 2009.

[16] Trusted Computing Group, *IF-M: TLV Binding*, Specification Version 1.0, March 2010.

[17] Sahita, R., Hanna, S., Hurst, R., and K. Narayan, "PB-TNC: A Posture Broker (PB) Protocol Compatible with Trusted Network Connect (TNC)", RFC 5793, March 2010.

[18] Sangster, P., Cam-Winget, N., and J. Salowey, “PT-TLS: A TCP-based Posture Transport (PT) Protocol”, RFC 6876, February 2013.

[19] Trusted Computing Group, *TNC IF-T: Binding to TLS,* Specification Version 2.0, February 2013.