[MC-PRCH]:
Peer Channel Protocol Specification

Revision Summary

Date	Revision History	Revision Class	Comments
08/10/2007	0.1	Major	Initial Availability
09/28/2007	0.2	Minor	Updated the technical content.
10/23/2007	0.3	Minor	Updated the technical content.
11/30/2007	0.3.1	Editorial	Revised and edited the technical content; updated links.

Date	Revision History	Revision Class	Comments
01/25/2008	0.3.2	Editorial	Revised and edited the technical content.

Table of Contents

1 Introduction

The Peer Channel Protocol is used for broadcasting messages over a virtual network of cooperating **nodes**. This protocol is used to send and receive messages between nodes in a named **mesh**. The nodes form the network by establishing connections to each other using a **discovery service** in which every node registers itself into a named mesh and **discovers** other nodes using the name of the mesh. The network is not fully connected. Instead, it is sparsely connected, yet a message sent by any node is propagated to the entire mesh by nodes forwarding to each other in a cooperative manner.

Each node forwards a message to all other **neighbors**. Each node is responsible for detecting and dropping duplicates of a message.

Each node maintains connections to a few other nodes in the mesh. A node must track the health of the **neighbor connection** and tune its neighbor set based on utility of the neighbor connection.

1.1 Glossary

Authenticator: A security token of a node computed using the password of the **mesh** and the node's public key.

Channel Type: A logical grouping of operations (messages) that can be sent over the **mesh**. A **mesh** can be used to handle more than one channel type simultaneously. A channel type is identified by a unique URI.

Discovery: Process used to discover other nodes in the **mesh** of interest.

Discovery Service: Service used to **discover** other nodes.

Endpoint: A tuple (composed of an IP address, port, and protocol number) that uniquely identifies a communication endpoint.

Flood (or Flooding): The process of propagating messages throughout a **mesh**.

Flood Message: Application message.

Mesh: A network of nodes that are all identified with the same **mesh name**.

Mesh Name: Identifies a set of **nodes** that establish connections to each other to form a **mesh**, as described in 1.3.1.

Neighbor: A node that is directly connected to the given node.

Neighbor Connection: A TCP/IP connection between two nodes' **endpoints**.

Node: An instance of a channel **endpoint** participating in the **mesh** that implements the Peer Channel Protocol.

Multi-Homing: Allowing TCP/IP connections on more than one interface adapter and network scope.

PeerChannel: The protocol detailed in this specification.

PNRP: Abbreviated form for Peer Name Resolution Protocol (PNRP) Version 4.0 (for more information, see [MS-PNRP]). Used with **PeerChannel** as a **discovery** mechanism.

Requesting Node: A node that is requesting the formation of a **neighbor connection** to another node in the **mesh**.

Responding Node: A node that is responding to a request to form a **neighbor connection** from another node in the **mesh**.

MAY, SHOULD, MUST, SHOULD NOT, MUST NOT: These terms (in all caps) are used as specified in [RFC2119]. All statements of optional behavior use either MAY, SHOULD, or SHOULD NOT.

1.2 References

1.2.1 Normative References

We conduct frequent surveys of the normative references to assure their continued availability. If you have any issue with finding a normative reference, please contact dochelp@microsoft.com. We will assist you in finding the relevant information. Please check the archive site, http://msdn2.microsoft.com/en-us/library/E4BD6494-06AD-4aed-9823-445E921C9624, as an additional source.

[MC-NBFS] Microsoft Corporation, ".NET Binary Format: SOAP Data Structure", July 2007.

[MC-NBFSE] Microsoft Corporation, ".NET Binary Format: SOAP Extension", July 2007.

[MC-NMF] Microsoft Corporation, ".NET Message Framing Protocol Specification", October 2007.

[MS-ERREF] Microsoft Corporation, "Windows Error Codes", January 2007.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997, http://www.ietf.org/rfc/rfc2119.txt

[RFC3484] Draves, R., "Default Address Selection for Internet Protocol version 6 (IPv6)", RFC 3484, February 2006, http://www.ietf.org/rfc/rfc3484.txt

[RFC4122] Leach, P., Mealling, M., and Salz, R., "A Universally Unique Identifier (UUID) URN Namespace", RFC 4122, July 2005, http://www.ietf.org/rfc/rfc4122.txt

[RFC4346] Dierks, T. and Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.1", RFC 4346, April 2006, http://www.ietf.org/rfc/rfc4346.txt

[SOAP1.2/1] Gudgin, M., Hadley, M., Mendelsohn, N., Moreau, J., and Nielsen, H.F., "SOAP Version 1.2 Part 1: Messaging Framework", W3C Recommendation, June 2003, http://www.w3.org/TR/2003/REC-soap12-part1-20030624

[SOAP1-2.1] Gudgin, M., Hadley, M., Mendelsohn, N., Moreau, J. J., Nielsen, H. F., Karmarkar, A. and Lafon, Y., " SOAP Version 1.2 Part 1: Messaging Framework (Second Edition) ", W3C Recommendation 27, April 2007, http://www.w3.org/TR/soap12-part1

[WSAddressing] Box, D. et. al, "Web Services Addressing (WS-Addressing)", August 2004, http://www.w3.org/Submission/ws-addressing/

If you have any trouble finding [WSAddressing], please check here.

[WSDL] Christensen, E., Curbera, F., Meredith, G., and Weerawarana, S., "Web Services Description Language (WSDL) 1.1", W3C Note, March 2001, http://www.w3.org/TR/2001/NOTE-wsdl-20010315

[WSTrust] IBM, Microsoft, Nortel, VeriSign, "WS-Trust V1.0", December 2005, http://www.oasis-open.org/committees/download.php/15980/oasis-wssx-ws-trust-1.0.pdf

1.2.2 Informative References

[MC-PRCR] Microsoft Corporation, "Peer Channel Custom Resolver Protocol Specification", July 2007

[MS-PNRP] Microsoft Corporation, "Peer Name Resolution Protocol (PNRP) Version 4.0 Specification", July 2007.

[MSDN-SECURITY_INFORMATION] Microsoft Corporation, "SECURITY_INFORMATION", http://msdn2.microsoft.com/en-us/library/aa379573.aspx

1.3 Protocol Overview (Synopsis)

Nodes using the Peer Channel Protocol create a mesh of redundant connections used for broadcasting and receiving messages in a decentralized manner. Messages sent by any node should typically reach all other nodes; the Peer Channel Protocol is not intended for sending point-to-point messages.

Nodes learn of other participating nodes in the mesh via a resolver service or referrals from existing neighbors. Each node uses this information to establish neighbor connections. Depending on the application configuration, these connections may or may not be secured.

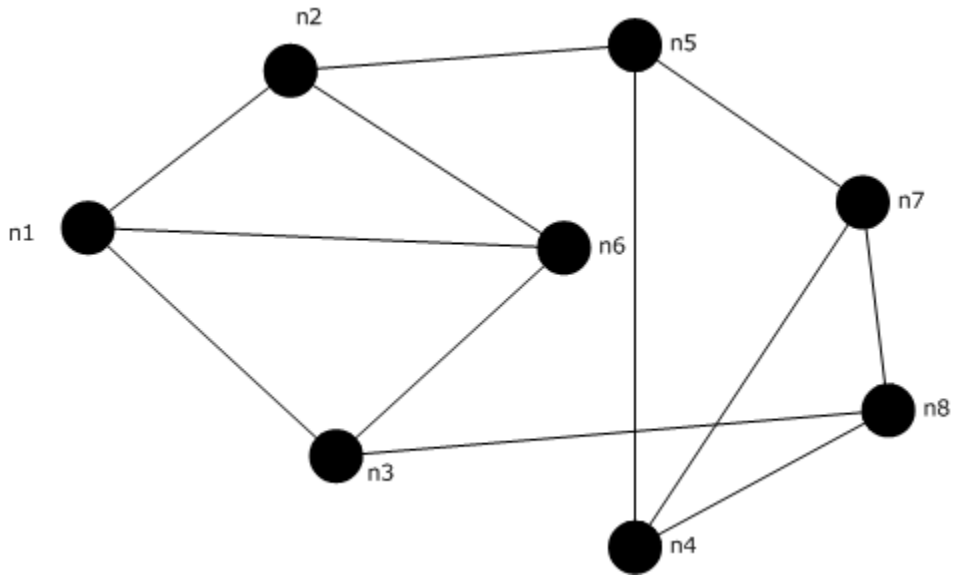

Figure 1: Sample diagram of a mesh

The above diagram shows one possible mesh shape with eight participating nodes. The mesh periodically reconfigures itself as the membership and message flow patterns change.

A mesh achieves broadcast semantics by means of sending messages to immediate neighbors who, in turn, forward the messages to their neighbors. This forwarding process stops when all participants in the mesh have received the message at least once.

1.3.1 Mesh and Mesh Names

A **mesh name** is used to identify a set of nodes that establish connections to each other to form a mesh. The name is any unique identifier that follows the host name syntax rules of URI. This name is used as the key to look up and resolve node **endpoints** in a discovery service.

Examples of valid mesh names follow:

- JoesDocumentUpdateNotice
- BobsNewsFlash
- AdamsStockTicker

1.3.2 ChannelTypes

A ChannelType is defined as a logical grouping of operations (messages) that can be sent over the mesh. A mesh can be used to handle more than one **channel type** simultaneously.

A channel type is identified by a unique URI. The HostName property of the URI must match the mesh name, and the scheme of the URI must be "net.p2p".

Some example channel type URIs in the mesh "BobsNewsFlash" follow:

- net.p2p://BobsNewsFlash/Political
- net.p2p://BobsNewsFlash/Financial/Stocks

1.3.3 Discovery

The Peer Channel Protocol uses a discovery service as a repository to store and retrieve each node's endpoint information. All nodes participating in a given mesh use the same discovery service. A node uses the discovery service to obtain connection information for a few nodes already present in the mesh to be joined. The node uses this to establish neighbor connections. The discovery service may return endpoints that are not currently active due to transient conditions. Connecting nodes can handle such error conditions by requesting additional connection information from the discovery service, and then retrying the connect operations.

1.3.4 Connecting to Other Nodes

A node typically establishes three neighbor connections, if possible. A node that does not discover other nodes initially will be initially alone but will be discovered by other nodes that join the mesh later. Nodes register (and update) their endpoint information in the discovery service for the duration of their participation in the mesh. Nodes also periodically tune neighbor connection sets by dropping the least useful connections and acquiring new connections. Usefulness of a connection is determined by the number of new messages received over that connection.

To establish a connection, the **requesting node** sends a message requesting a connection from another node. The **responding node** sends back a message indicating its availability. If the connection is accepted by the responding node, the connection is now ready for sending and receiving application messages.

1.3.5 Exchanging Application Messages

After establishing connections with one or more neighbors, a node is ready to send and receive application messages. If per-message security is configured, each message is first processed for

security before processing. All application messages received are forwarded to all connected neighbors except to the neighbor from whom the message is received.

All nodes receive all messages addressed to the mesh even if some of the messages are only intended for a subset of the mesh.

Each message is identified by a unique message ID that is generated by the node that initially creates the message. Because a particular node may receive a message multiple times as a result of having multiple neighbors, this message ID is used to detect and discard all duplicate messages.

Outgoing messages (called **flood messages**) are created by adding a Peer Channel Protocol header to the message (see section 2.2.2.7), and then sending the messages to the corresponding ChannelType URI.

1.3.6 Security

A mesh can be secured at neighbor transport layer, message layer, or both.

1.3.6.1 Transport-Layer Security

A mesh may be configured to send and receive all messages over a secure transport. In this case, all neighbor-to-neighbor connections established will be TLS over TCP connections, as specified in [RFC4346]. Applications can use two different types of credentials for achieving transport-layer security.

1.3.6.1.1 Password

Every node that attempts to join the mesh is required to prove knowledge of the mesh password. A secure neighbor-to-neighbor connection is established using any arbitrary X509 certificate (this certificate does not need to be trusted). A message exchange takes place in which both nodes exchange messages to send tokens that prove their knowledge of the password. Each node validates the other node's security token before initiating further message exchanges with that node.

1.3.6.1.2 Trusted Certificate

Every node has a certificate that all nodes can validate and trust that is provisioned out of band. Secure neighbor-to-neighbor connections are established using these certificates. Applications provide these certificates and the associated authentication scheme. The validation scheme has to be generic to include any node's certificate (it is unpredictable what other nodes a given node will be connected to at any time, so all nodes have to implement generic authentication schemes). However, no message exchange takes place. If the nodes fail to authenticate each other's certificate, the neighbor connection is dropped.

1.3.6.2 Message-Layer Security

Independent of transport-layer security, the Peer Channel Protocol also supports per-message security. Application messages (not protocol messages) are signed with a trusted X509 certificate to make individual messages tamper-proof. The application must provide the scheme to validate and trust the certificate used to secure the message. The vendor must also distribute the certificates used to validate these messages.

1.4 Relationship to Other Protocols

The Peer Channel Protocol depends on the following non-native protocols:

- .Net TLSNego Protocol Specification, as specified in [RFC4346]: For establishing TLS-based secure TCP endpoints.
- .NET Message Framing Protocol Specification, as specified in [MC-NMF]: For exchanging encoded SOAP messages over TCP.
- .NET Binary Format: SOAP Data Structure, as specified in [MC-NBFS]: For exchanging a compactly encoded stream of data between two nodes.
- .NET Binary Format: SOAP Extension, as specified in [MC-NBFSE]: For exchanging strings once, and then referring to them in subsequent documents.

1.5 Prerequisites/Preconditions

It is assumed that a node connecting to the mesh is configured with the following details:

- Mesh name.
- Connection information for discovery service.
- Security mechanism employed in the mesh, and the credentials needed to authenticate into the mesh.
- Credentials needed to sign flood messages in case the message authentication feature is used.
- URI of each of the channels that it wants to send messages to (or receive messages from).

It is assumed that these details are available to all participating nodes before connecting to the mesh. The Peer Channel Protocol is not used to communicate these details.

1.6 Applicability Statement

The Peer Channel Protocol is suitable for scenarios in which messages sent by any node should reach all other nodes participating in a single named mesh. The Peer Channel Protocol is not intended for sending point-to-point messages in a mesh. All messages must be addressed to the mesh, not to any particular peer.

The Peer Channel Protocol is suited for use in scenarios that do not require a high degree of reliability because it does not include any mechanism to guarantee message delivery.

1.7 Versioning and Capability Negotiation

The Peer Channel Protocol does not have a version negotiation scheme.

1.8 Vendor-Extensible Fields

The Peer Channel Protocol does not have any vendor-extensible fields.

1.9 Standards Assignments

No standards assignments have been made for this protocol.

2 Messages

2.1 Transport

A node configured without transport security MUST use TCP as the neighbor-to-neighbor transport. A node configured with transport security MUST use TLS to secure the channel, as specified in [RFC4346].

2.2 Message Syntax

The Peer Channel Protocol is comprised of messages that are based on SOAP (as specified in [SOAP1.2/1]) syntax. Peer Channel Protocol messages are defined as a WSDL (as specified in [WSDL]) operation binding. Peer Channel Protocol messages define the Action header and the element type in the SOAP body with the exception of FloodMessages, which are identified by the presence of other Peer Channel Protocol-specific headers in the SOAP message.

2.2.1 Peer Channel Protocol Structures

Peer Channel Protocol-specific structures are specified in this section. These structures are reused across several Peer Channel Protocol messages.

2.2.1.1 PeerHashToken Element

The PeerHashToken element contains a node's **authenticator** token. for details on how the PeerHashToken is computed using a node's certificate and the mesh password, see section 2.2.2.1.1.

```
<xs:schema xmlns:tns="http://schemas.microsoft.com/net/2006/05/peer" a
ttributeFormDefault="unqualified" elementFormDefault="qualified" targe
tNamespace="http://schemas.microsoft.com/net/2006/05/peer" xmlns:xs="h
ttp://www.w3.org/2001/XMLSchema">
  <xs:element name="PeerHashToken">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="Authenticator" type="xs:base64Binary" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

ElementName	Description
PeerHashToken	MUST contain the token being validated.
PeerHashToken/Authenticator	MUST contain a token in base64-encoded form.

2.2.1.2 PeerNodeAddress Structure

The PeerNodeAddress structure contains the URI of the node and the set of IPAddresses on which the node is listening.

```
<xs:schema
```

```
  xmlns:tns="http://schemas.datacontract.org/2004/07/System.Net"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://schemas.datacontract.org/2004/07/System.Net"
  xmlns:a="http://www.w3.org/2005/08/addressing/"
  attributeFormDefault="unqualified"
  elementFormDefault="qualified" >
  <xs:import namespace="http://schemas.microsoft.com/2003/10/Serializa
tion/Arrays" />
  <xs:import namespace="http://www.w3.org/2005/08/addressing/" />
    <xs:complexType name="IPAddress">
      <xs:sequence>
        <xs:element name="m_Address" type="xs:unsignedInt" />
        <xs:element name="m_Family" type="xs:string" />
        <xs:element name="m_HashCode" type="xs:unsignedInt" />
        <xs:element name="m_Numbers">
          <xs:complexType>
            <xs:sequence>
              <xs:element maxOccurs="unbounded" xmlns:q1="http://schem
as.microsoft.com/2003/10/Serialization/Arrays" ref="q1:unsignedShort"
/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
        <xs:element name="m_ScopeId" type="xs:unsignedInt" />
      </xs:sequence>
    </xs:complexType>
    <xs:complexType name="PeerNodeAddress">
      <xs:sequence>
        <xs:element name="EndpointAddress">
          <xs:complexType>
            <xs:sequence>
              <xs:element ref="a:Address" />
            </xs:sequence>
          </xs:complexType>
        </xs:element>
        <xs:element name="IPAddresses">
          <xs:complexType>
            <xs:sequence>
              <xs:element maxOccurs="unbounded" name="IPAddress" />
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
</xs:schema>
```

ElementName	Description
Address	MUST reference the node's endpoint as a URI.
IPAddresses	MUST contain one or more IPAddress structures.
IPAddress	Describes a complete IPAddress.
IPAddress/m_Address	"0" MUST be used to indicate an IPv6 address. Otherwise, it MUST contain an address as an unsigned 32-bit number.

ElementName	Description
IPAddress/m_Family	Address family of the IPAddress. The value MUST be "Internetwork" if the address is an IPv4 address, or "InternetworkV6" if the address is an IPv6 address.
IPAddress/m_HashCode	This value SHOULD be set to "0". On parsing this field from a received message, this element MUST be ignored.<1>
IPAddress/m_Numbers	This element MUST contain the serialized version of the address bytes grouped as 16-bit numbers in big-endian format. For IPv4 addresses, this element SHOULD contain 0 instances. For IPv6 addresses, this element MUST contain exactly 8 "unsignedShort" sub-elements.
IPAddress/m_Numbers/unsignedShort	MUST contain a 16-bit number.
IPAddress/m_ScopeId	For IPv6 address, this element MUST contain the Scope ID of the address. For IPv4 addresses, this element MUST be ignored.

2.2.1.3 Referral Structure

A Referral contains a node's endpoint information. For how Referrals are used, see section 3.1. Note that the Referral structure itself does not include any information on the node that is sending or receiving the Referral; it only contains information on the referred node.

```
<xs:schema xmlns:d7p1="http://schemas.microsoft.com/2003/10/Serializat
ion/Arrays" xmlns:a="http://www.w3.org/2005/08/addressing" xmlns:d5p1=
"http://schemas.datacontract.org/2004/07/System.Net" attributeFormDefa
ult="unqualified" elementFormDefault="qualified" xmlns:xs="http://www.
w3.org/2001/XMLSchema">
  <xs:import namespace="http://www.w3.org/2005/08/addressing" />
  <xs:import namespace="http://schemas.xmlsoap.org/ws/2006/02/addressi
ngidentity" />
  <xs:import namespace="http://schemas.datacontract.org/2004/07/System
.Net" />
  <xs:complexType name="Referral">
      <xs:sequence>
        <xs:element name="NodeId" type="xs:unsignedLong" />
        <xs:element name="Address" type="PeerNodeAddress" />
      </xs:sequence>
  </xs:complexType>
  <xs:complexType name="Referrals">
    <xs:sequence maxOccurs="unbounded" >
      <xs:element name ="Referral"/>
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

ElementName	Description
Referral	Information identifying a single node in the mesh.
Referral/NodeId	MUST contain a 64-bit unique identifier.

ElementName	Description
Referral/Address	MUST contain the PeerNodeAddress of the node.

2.2.1.4 RefuseReason Enumeration

The **RefuseReason** enumeration describes the reason a requested neighbor connection has been denied.

```
typedef  enum
{
  DuplicateNeighbor,
  DuplicateNodeId,
  NodeBusy
} RefuseReason;
```

DuplicateNeighbor: A connection request by a node is refused because a connection between the two nodes already exists. A connection is deemed duplicate if the **GUID** part of the listen URI of the PeerNode matches.

DuplicateNodeId: The responding node already has a connection to a node with the same **NodeId** as the **NodeId** given in the corresponding Connect message.

NodeBusy: The responding node has already connected to the configured maximum number of nodes.

2.2.1.5 DisconnectReason Enumeration

The **DisconnectReason** enumeration describes the reason a neighbor connection is closed.

```
typedef  enum
{
  LeavingMesh,
  NotUsefulNeighbor,
  DuplicateNeighbor,
  DuplicateNodeId
} DisconnectReason;
```

LeavingMesh: The disconnecting node is leaving the mesh.

NotUsefulNeighbor: The receiving node that is receiving the message has been determined to be less useful than other neighbor nodes, as given by the LinkUtilityIndex. For how the LinkUtility is calculated, see section 2.2.2.8.

DuplicateNeighbor: A connection to the receiving node already exists.

DuplicateNodeId: A connection to a node with the same **NodeId** as the receiving node already exists.

2.2.1.6 FloodMessage Header

The FloodMessage header is used to identify flood (application) messages. The header MUST be formatted as follows.

```
<p:FloodMessage xmlns:p="http://schemas.microsoft.com/net/2006/05/peer
">
PeerFlooder
</p:FloodMessage>
```

2.2.1.7 Endpoint Format

An endpoint URI has the following syntax:

Scheme://Host:Port/Path/Guid

Where:

- **Scheme**: "net.p2p" for neighbor connections or "net.tcp" for a connection with a resolver service.
- **Host**: Host name or IPAddress associated with the host on which the node is created.
- **Port**: Configured port for the node's endpoint.
- **Path**: URI's path.
- **Guid**: Generated and assigned to the node's endpoint. The GUID MUST be formatted as specified in [RFC4122].

2.2.2 Peer Channel Protocol Messages

2.2.2.1 RequestSecurityToken Message

The RequestSecurityToken (RST) message syntax is as specified in [WSTrust]. The PeerHashToken element is used as the CustomToken binding of this message. Refer to [WSDL] for a description of WSDL schema and the meaning of [WSDL] specific xml elements.

```
<wsdl:operation xmlns:wsdl="http://schemas.xmlsoap.org/wsdl" name="Req
uestSecurityToken" xmlns:wsa10="http://schemas.xmlsoap.org/ws/2004/08/
addressing">
      <wsdl:input wsa10:Action="RequestSecurityToken" name="RequestSec
urityToken" message="RequestSecurityToken" />
<wsdl:output wsa10:Action="RequestSecurityTokenResponse" name="Request
SecurityTokenResponse" message="RequestSecurityTokenResponse" />
    </wsdl:operation>
```

ElementName	Legal value
RequestSecurityToken/TokenType	"http://schemas.microsoft.com/net/2006/05/peer/peerhashtoken"
RequestSecurityToken/RequestType	"http://schemas.xmlsoap.org/ws/2005/02/trust/Validate"

2.2.2.1.1 Computing the PeerHashToken

The PeerHashToken only contains a PeerAuthenticator element. The authenticator element carries a base64-encoded security token as the text node. The security token is an HMACSHA256 value that MUST be computed as follows.

NodeSecurityToken = **HMACSHA256**(HASHEDKEY)
HASHEDKEY = (**SHA256**(**PWD**)+**PUBLICKEY**)

Where:

- **HMACSHA256** is the HMAC function with hash function SHA256.
- **SHA256** refers to the SHA256 hash algorithm.
- **PWD** is the password as a Unicode byte stream. PWD bytes are used as the secret for the HMACSHA256 function.
- **PUBLICKEY** is the public key of the node for which the PeerHashToken is being computed. Public key bits of the certificate that are provisioned for the neighbor connection MUST be used here.
- **HASHEDKEY** is computed by concatenating the byte streams of (a) the output of the function SHA256 over the PWD and (b) the public key in the node's certificate.

2.2.2.2 RequestSecurityTokenResponse Message

The RequestSecurityTokenResponse message carries the validation results of the requesting node's PeerHashToken element by the responding node. It also contains the PeerHashToken of the responding node. The schema of this message is contained in [WSTrust] section 4.2.

ElementName	Legal value
RequestSecurityTokenResponse/TokenType	MUST contain the URI "http://schemas.microsoft.com/net/2006/05/peer/peerhashtoken".
RequestSecurityTokenResponse/Status	MUST contain an instance of "http://schemas.xmlsoap.org/ws/2005/02/trust/Code" element.
RequestSecurityTokenResponse/Status/Code	MUST have the URI "http://schemas.xmlsoap.org/ws/2005/02/trust/status/valid" as the text node. In case the recipient is not able to validate the token in the incoming message, the connection must be aborted.
RequestSecurityTokenResponse/RequestedSecurityToken	MUST contain an instance of PeerHashToken containing the hash of the responding party. For how to compute the hash, see section 2.2.2.1.1.

2.2.2.3 Connect Message

The Connect message is used to request a connection to another node.

```
<xs:schema xmlns:tns="http://schemas.microsoft.com/net/2006/05/peer" a
```

```
ttributeFormDefault="unqualified" elementFormDefault="qualified" targe
tNamespace="http://schemas.microsoft.com/net/2006/05/peer" xmlns:xs="h
ttp://www.w3.org/2001/XMLSchema">
<xs:complexType name="Connect">
      <xs:sequence>
        <xs:element ref="PeerNodeAddress" />
        <xs:element name="NodeId" type="xs:unsignedLong" />
      </xs:sequence>
    </xs:complexType>
</xs:schema>

<wsdl:operation name="Connect" xmlns:wsdl="http://schemas.xmlsoap.org/
wsdl/" xmlns:wsa10="http://schemas.xmlsoap.org/ws/2004/08/addressing">
      <wsdl:input wsa10:Action="http://schemas.microsoft.com/net/2006/
05/peer/Connect" name="Connect" message="Connect" />
    </wsdl:operation>
```

ElementName	Legal value
Connect	Requests a neighbor connection. MUST only contain information pertaining to the requesting node.
Connect/Address	MUST contain the PeerNodeAddress of the requesting node.
Connect/NodeId	MUST contain the NodeId of the requesting node.

2.2.2.4 Welcome Message

The Welcome message is sent by a responding node to accept a neighbor connection.

```
<xs:schema xmlns:tns="http://schemas.microsoft.com/net/2006/05/peer" a
ttributeFormDefault="unqualified" elementFormDefault="qualified" targe
tNamespace="http://schemas.microsoft.com/net/2006/05/peer" xmlns:xs="h
ttp://www.w3.org/2001/XMLSchema">
<xs:complexType name="Welcome">
      <xs:sequence>
        <xs:element name="NodeId" type="xs:unsignedLong" />
        <xs:element name="Referrals" type="Referrals" />
      </xs:sequence>
    </xs:complexType>
</xs:schema>

<wsdl:operation xmlns:wsdl="http://schemas.xmlsoap.org/wsdl" name="Wel
come" xmlns:wsa10="http://schemas.xmlsoap.org/ws/2004/08/addressing">
      <wsdl:input wsa10:Action="http://schemas.microsoft.com/net/2006/
05/peer/Welcome" name="Welcome" message="Welcome" />
    </wsdl:operation>
```

ElementName	Description
Welcome	Indicates to the requesting node that a connection request has been accepted.
Welcome/NodeId	MUST contain the NodeId of the responding node.

ElementName	Description
Welcome/Referrals	A collection of Referral elements. Each element in the Referrals collection MUST refer to a neighbor to which the responding node is currently connected.

2.2.2.5 Refuse Message

The Refuse message is sent by a responding node to reject a neighbor connection.

```
<xs:schema xmlns:tns="http://schemas.microsoft.com/net/2006/05/peer" a
ttributeFormDefault="unqualified" elementFormDefault="qualified" targe
tNamespace="http://schemas.microsoft.com/net/2006/05/peer" xmlns:xs="h
ttp://www.w3.org/2001/XMLSchema">
<xs:complexType name="Refuse">
      <xs:sequence>
        <xs:element name="Reason" />
    <xs:restriction base ="xs:string">
      <xs:enumeration value ="DuplicateNeighbor"/>
      <xs:enumeration value ="DuplicateNodeId"/>
      <xs:enumeration value ="NodeBusy"/>
    </xs:restriction>
        <xs:element name="Referrals" type="Referrals" />
      </xs:sequence>
    </xs:complexType>
</xs:schema>

<wsdl:operation xmlns:wsdl="http://schemas.xmlsoap.org/wsdl" xmlns:wsa
10="http://schemas.xmlsoap.org/ws/2004/08/addressing" name="Refuse">
      <wsdl:input wsa10:Action="http://schemas.microsoft.com/net/2006/
05/peer/Refuse" name="Refuse" message="Refuse" />
    </wsdl:operation>
```

ElementName	Description
Refuse	Indicates to the requesting node that the connection request has been denied.
Refuse/Reason	MUST contain a valid **RefuseReason** enumeration value indicating the error causing the denial of the neighbor connection.
Refuse/Referrals	A collection of Referral elements. Each element in the Referrals collection MUST refer to a node to which the responding neighbor is currently connected.

2.2.2.6 Disconnect Message

The Disconnect message is sent by a node to close a neighbor connection.

```
<xs:schema xmlns:tns="http://schemas.microsoft.com/net/2006/05/peer" a
ttributeFormDefault="unqualified" elementFormDefault="qualified" targe
tNamespace="http://schemas.microsoft.com/net/2006/05/peer" xmlns:xs="h
ttp://www.w3.org/2001/XMLSchema">
<xs:complexType name="Disconnect">
      <xs:sequence>
```

```
          <xs:element name="Reason" >
           <xs:restriction base ="xs:string">
             <xs:enumeration value ="LeavingMesh"/>
             <xs:enumeration value ="NotUsefulNeighbor"/>
             <xs:enumeration value ="DuplicateNeighbor"/>
             <xs:enumeration value ="DuplicateNodeId"/>
             <xs:enumeration value ="NodeBusy"/>
           </xs:restriction>
          </xs:element>
          <xs:element name="Referrals" type="Referrals" />
        </xs:sequence>
      </xs:complexType>
</xs:schema>

<wsdl:operation xmlns:wsdl="http://schemas.xmlsoap.org/wsdl" xmlns:wsa
10="http://schemas.xmlsoap.org/ws/2004/08/addressing" name="Disconnect
">
      <wsdl:input wsa10:Action="http://schemas.microsoft.com/net/2006/
05/peer/Disconnect" name="Disconnect" message="Disconnect" />
    </wsdl:operation>
```

ElementName	Legal value
Disconnect	Indicates to the node receiving this message that the connection between the sending and receiving nodes is being shut down.
Disconnect/Reason	MUST contain a valid **DisconnectReason** enumeration value indicating the cause for disconnecting the neighbor connection.
Disconnect/Referrals	A collection of Referral elements. Each element in the Referrals collection MUST refer to a node to which the responding neighbor is currently connected.

2.2.2.7 Flood (Application) Message

The Flood (Application) message contains application-specific information.

All flood messages MUST add the following headers (in the namespace "http://schemas.microsoft.com/net/2006/05/peer"):

Name	Description
MessageID	GUID that MUST uniquely identify the message in the mesh.
FloodMessage	MUST be a valid FloodMessage header.
PeerVia	Identifies the target ChannelType of the message. MUST contain the URI of the node's listening endpoint.
PeerTo	Identifies the specific target for the message. SHOULD be set to the same value as **PeerVia**.

2.2.2.8 LinkUtility Message

The LinkUtility message is used to transmit the LinkUtility value to another neighbor, indicating the usefulness of their neighbor connection.

```
<xs:schema xmlns:tns="http://schemas.microsoft.com/net/2006/05/peer" a
ttributeFormDefault="unqualified" elementFormDefault="qualified" targe
tNamespace="http://schemas.microsoft.com/net/2006/05/peer" xmlns:xs="h
ttp://www.w3.org/2001/XMLSchema">
<xs:complexType name="LinkUtilityInfo">
    <xs:sequence>
      <xs:element minOccurs="1" maxOccurs="1" name="Total" type="xs:un
signedInt" />
      <xs:element minOccurs="1" maxOccurs="1" name="Useful" type="xs:u
nsignedInt" />
    </xs:sequence>
  </xs:complexType>
</xs:schema>

<wsdl:operation xmlns:wsdl="http://schemas.xmlsoap.org/wsdl" xmlns:wsa
10="http://schemas.xmlsoap.org/ws/2004/08/addressing" name="LinkUtilit
y">
      <wsdl:input wsa10:Action="http://schemas.microsoft.com/net/2006/
05/peer/LinkUtility" message="LinkUtilityInfo" />
    </wsdl:operation>
```

Element name	Description
LinkUtilityInfo	MUST contain LinkUtilityInfo details.
LinkUtilityInfo/Total	MUST contain the total number of messages received by the node since the last LinkUtilityInfo message. MUST NOT refer to a cumulative total.
LinkUtilityInfo/Useful	MUST indicate the number of messages (out of the total) that were not duplicates.

2.2.2.9 Ping Message

The Ping message is used to check the validity of a connection when a node resumes activity from standby. It MUST contain no body.

```
<wsdl:operation xmlns:wsdl="http://schemas.xmlsoap.org/wsdl" xmlns:wsa
10="http://schemas.xmlsoap.org/ws/2004/08/addressing" name="Ping">
   <wsdl:input wsa10:Action="http://schemas.microsoft.com/net/2006/05/
peer/Ping" name="Ping" />
</wsdl:operation>
```

3 Protocol Details

The Peer Channel Protocol will be defined from the perspective of two distinct roles:

- The receiving node: Processes incoming messages and connection requests.
- The sending node: Transmits outbound application messages to neighbors.

All nodes implementing the Peer Channel Protocol MUST implement both roles.

3.1 Receiving Node Details

3.1.1 Abstract Data Model

This section describes a conceptual model of a possible data organization that an implementation maintains to participate in this protocol. The described organization is provided to facilitate the explanation of how the protocol behaves. This document does not mandate that implementations adhere to this model as long as their external behavior is consistent with what is described in this document.

The receiving node MUST store the following information:

- **Local Endpoint Information**: The node MUST store its listening endpoint addresses. This information SHOULD be published as a PeerNodeAddress into the discovery service under the mesh name corresponding to the mesh name used in the application. For more information, see [MC-PRCR].
- **Mesh Name**: The node MUST store locally the string value of the mesh name for use in interacting with the resolver service.
- **NodeId**: This is an eight-byte unsigned number that is randomly generated on creation of the node itself.
- **MessageId Cache**: Each node MUST maintain a cache of previously seen **MessageIds**. This is used to detect duplicate messages. A node SHOULD cache **MessageIds** for at least five minutes.
- **ChannelType Information**: The following information MUST be stored for each ChannelType supported by the receiving node.
  - ChannelUri: The URI to which the ChannelType corresponds. This value MUST match the PeerVia header in the incoming message.
  - MessageAuthentication Callback: The callback that is invoked to verify the incoming message signature if the particular ChannelType supports message authentication.
  - MessageDispatcher Callback: The callback that accepts incoming messages for processing. Local processing of this message is handed off to this callback by the node.
- **(Optional) X.509 Certificate for Transport Security**: An X509 certificate for the key that is used to establish TLS over TCP connections. Only required if certificates are used to secure mesh connections.
- **(Optional) X.509 Certificate for Message Signing**: An X509 certificate for the key that is used to sign messages. Only needed if message authentication is enabled on mesh messages.
- **(Optional) Password**: Password that is used in security handshakes. See RequestSecurityToken message. Only needed if passwords are used to secure the mesh.

- **Discovery Service Information**: Information used to connect to the discovery service. This MUST include the service location, port number, transport, and any applicable security settings.
- **Neighbor Connection Information**: Details on neighbor connections that are already established. This is used to forward messages to other nodes in the mesh. The node MUST store the following information for each connected neighbor.
  - Neighbor Proxy Information: Endpoint information and connection handle (socket).
  - Neighbor Connection State: Created, Authenticated, Connected, or Closed.
  - LinkUtility Message Timer: Timer used to initiate a LinkUtility message.
  - Initiator: Flag indicating who initiated the connection. TRUE if the local node initiated the neighbor connection; otherwise, it's FALSE.
  - NodeId: Unsigned long number as reported by the remote node.

3.1.2 Timers

Each receiving node MUST have the following timers.

- Maintenance Timer:
  - This timer SHOULD<2> be triggered immediately when opening a node.
  - This timer is used to regularly run the maintenance cycle, which examines the neighbor connection set, and tunes it for optimal throughput. The duration SHOULD<3> be five minutes.
  - The timer SHOULD<4> be triggered immediately if the number of connected neighbors falls below MinConnectedNeighbors.
- MessageID Cache Timer: A periodic timer used to initiate MessageId cache maintenance. (This removes previously seen MessageIds to maintain a reasonable cache size.) The period SHOULD be one minute.

If the mesh is secured with a password, the following timer is created for each new connection.

- Security Handshake Timer: Used to close the connection if the remote neighbor does not send a timely response during the authentication protocol. The period SHOULD be one minute.

For each connection that is in Authenticated state, the following timer is created.

- Connect Timer: This timer exists to close the connection if the remote neighbor does not send a timely response. The period SHOULD be one minute.

For each connection that is in Connected state, the following timer is created.

- LinkUtility Timer: This timer exists for each neighbor connection to send a LinkUtility message at regular intervals. The period SHOULD be one minute.

3.1.3 Initialization

3.1.3.1 Setting Configuration

A node must be configured with the following information before connecting to a mesh.

- ListenIPAddress: An IPV4 or IPV6 address valid for the node that will be used to accept connection requests. If no ListenIPAddress is specified, the application is requesting support for **multi-homing**, and the node SHOULD<5> accept connection requests on all active network interfaces.
- Port: Port number on which the local node's TCP listener is opened. This information is published in the discovery service that is used by other nodes in the mesh to connect to the local node.
- Mesh Name: This is also passed to the discovery service to find other nodes in the mesh.
- Discovery Service Connection Information: This is used to obtain endpoint information of other nodes in the mesh.
- ChannelType Information: ChannelType definitions that the node will handle. There must be at least one ChannelType definition provided for the node to receive and send messages. For each channel type, the configuration must be provided.
  - ChannelType URI: A properly formatted ChannelType URI.
  - MessageDispatcher Callback: A callback function that processes the messages.
  - MessageValidator Callback: The message security verification callback.
- Security Mode and Security Configuration: The node must have all necessary security information to connect to the mesh if the mesh is configured to support security. All nodes participating in a single mesh MUST have the same security configuration.

3.1.4 Higher-Layer Triggered Events

A node MUST provide (to higher-layer applications and protocols) three logical operations that can be invoked.

- Opening a node (section 3.1.4.1).
- Receiving a message (section 3.1.4.2).
- Closing a node (section 3.1.4.3).

3.1.4.1 Opening a Node

When a higher-layer application or protocol triggers the Open event, the node MUST carry out the following procedure.

1. The node MUST create a TCP endpoint for accepting node connection requests. It MUST be created on the configured ListenIPAddress and port (see section 3.1.3.1).
2. The node MUST create a PeerNodeAddress structure to describe the node endpoint. It MUST be created using the ChannelType URI and configured ListenIPAddress for the node.
3. The node SHOULD initiate InitialMaintenance to establish connectivity with other nodes.
4. The node MUST publish the PeerNodeAddress in the discovery service using the mesh name configured (see section 3.1.3.1) by the application.
5. The node MUST monitor changes in the configured ListenIPAddresses and update the discovery service immediately following address change notifications.

3.1.4.2 Receiving a Message

If the mesh configuration requires that messages be signed, the receiver MUST look for the signature, and then verifies it. The node MUST forward the message to all of its neighbors (except the node that sent the message) unless the message is a duplicate of a previously received message, in which case the node MUST NOT forward the message.

Because all nodes act as message senders and receivers, a node SHOULD send a LinkUtility message to all of its neighbors from which it receives messages. Likewise, it SHOULD also receive a LinkUtility message from all of its connected neighbors to which it sends messages.

For each incoming message, a LinkUtilityIndex MUST be updated. A LinkUtility message is only sent if conditions are met. The **Useful** and **Total** values in the LinkUtility message MUST be updated on message reception to reflect the current state of the link. However, the LinkUtilityIndex MUST reflect cumulative values, and MUST never be reset once a neighbor connection is established.

The receiver of a LinkUtility message MUST check the values of **Useful** and **Total** in the message to make sure that they are within the bounds of unacknowledged message count.

3.1.4.3 Closing a Node

A node SHOULD take the following steps when closing down.

1. The node SHOULD remove its endpoint publication in the discovery service.
2. All neighbor connections SHOULD be closed by sending a Disconnect message to each neighbor with the **Reason** element set to "LeavingMesh".
3. The node SHOULD close any open endpoints.

3.1.5 Message Processing and Sequencing Rules

3.1.5.1 Receiving a RequestSecurityToken Message

The receiving node MUST follow the following sequence of rules for processing this message.

1. If the connection is not in Created state, the node MUST abort the neighbor connection and stop the protocol.
2. If the mesh is not configured to use password-based authentication, the receiving node MUST abort the neighbor connection and terminate the protocol.
3. The receiving node MUST compute the requesting node's Authenticator token using the password and the requesting node's public key. The requesting node's public key is available to the responding node as a result of TLS over TCP.
4. If the byte-wise comparison of computed token and the token retrieved from the message in the PeerHashToken **Authenticator** element do not match, the node MUST abort the connection and terminate the protocol.
5. The receiving node MUST send a RequestSecurityTokenResponse message (to the requesting node) that contains the status of the validation and the responding node's Authenticator token computed using its password and the responding node's public key.
6. The receiving node transitions the neighbor connection to the Authenticated state and starts the Connect Handshake timer.

In case of failures of any kind (communication, timing, security token validation), both nodes MUST drop the neighbor connection.

3.1.5.2 Receiving a RequestSecurityTokenResponse Message

The receiving node MUST follow the following sequence of rules for processing this message.

1. If the connection is not in Created state, or the node is not the initiator of the connection, the node MUST abort the connection and stop the protocol. This message MUST only be received as a response to a RequestSecurityToken message sent by the initiator of the neighbor connection immediately after establishing the connection.

2. Verify that the result of security token validation is success. If the validation token is not properly formed (see section 2.2.2.2), the receiving node MUST abort the connection and stop the protocol.

3. The receiving node MUST retrieve the Authenticator token (contained as a base64-encoded value in the Authenticator element at the path Envelope/Body/RequestSecurityTokenResponse/RequestedSecurityToken/PeerHashToken in the message).

4. The receiving node MUST compute the sender's Authenticator token using the sender's public key and the password.

5. The receiving node compares the Authenticator tokens computed in steps 3 and 4. If the byte-wise comparison of these Authenticator tokens fails, the receiving node MUST abort the connection and stop the protocol.

6. The receiving node MUST transition the connection to Authenticated state.

7. The receiving node SHOULD start the Connect Handshake timer.

3.1.5.3 Receiving a Connect Message

1. If the node is not in the Authenticated state, it MUST abort the neighbor connection and stop the protocol.

2. Validate that the Connect message contains a valid PeerNodeAddress. The **NodeId** value in the message MUST be nonzero. If any of the above validation fails, the connection MUST be aborted and the protocol terminated.

3. If the message contains a value in the **NodeId** that is equal to the receiver's **NodeId**, the node MUST send back a Refuse message with RefuseReason set to DuplicateNodeId.

4. If there is another connection to a node with the same **NodeId**, the node MUST send a Ping message on the existing connection. If the Ping message succeeds (that is, there are two different connections to the same node), a candidate connection to be dropped MUST be picked as follows.

   - If both connections are initiated by the same node, the newly established connection MUST be dropped. A Refuse message with RefuseReason set to DuplicateNeighbor MUST be sent on the new connection, and the new connection MUST be closed.

   - Otherwise, a connection that is initiated by a node that has a higher NodeId value is picked for closing (this can be either the requesting node or the responding node). If this is the new connection, a Refuse message MUST be sent with DuplicateNeighbor as the RefuseReason, and the connection MUST be closed. If this is the existing connection, a Disconnect message

MUST be sent with DuplicateNeighbor as the DisconnectReason, and the old connection MUST be closed.

5. If the new connection is not closed, a Welcome message MUST be sent with the receiving node's **NodeId** and referral set. The referral set MUST only consist of PeerNodeAddress structures corresponding to the receiving node's currently-connected neighbors.

6. If a Welcome message was sent to the requesting neighbor, the connection MUST transition to the Connected state. Both nodes can now exchange flood messages on the connection.

7. If a Welcome message was not sent as a response, the connection MUST be closed and the protocol terminated.

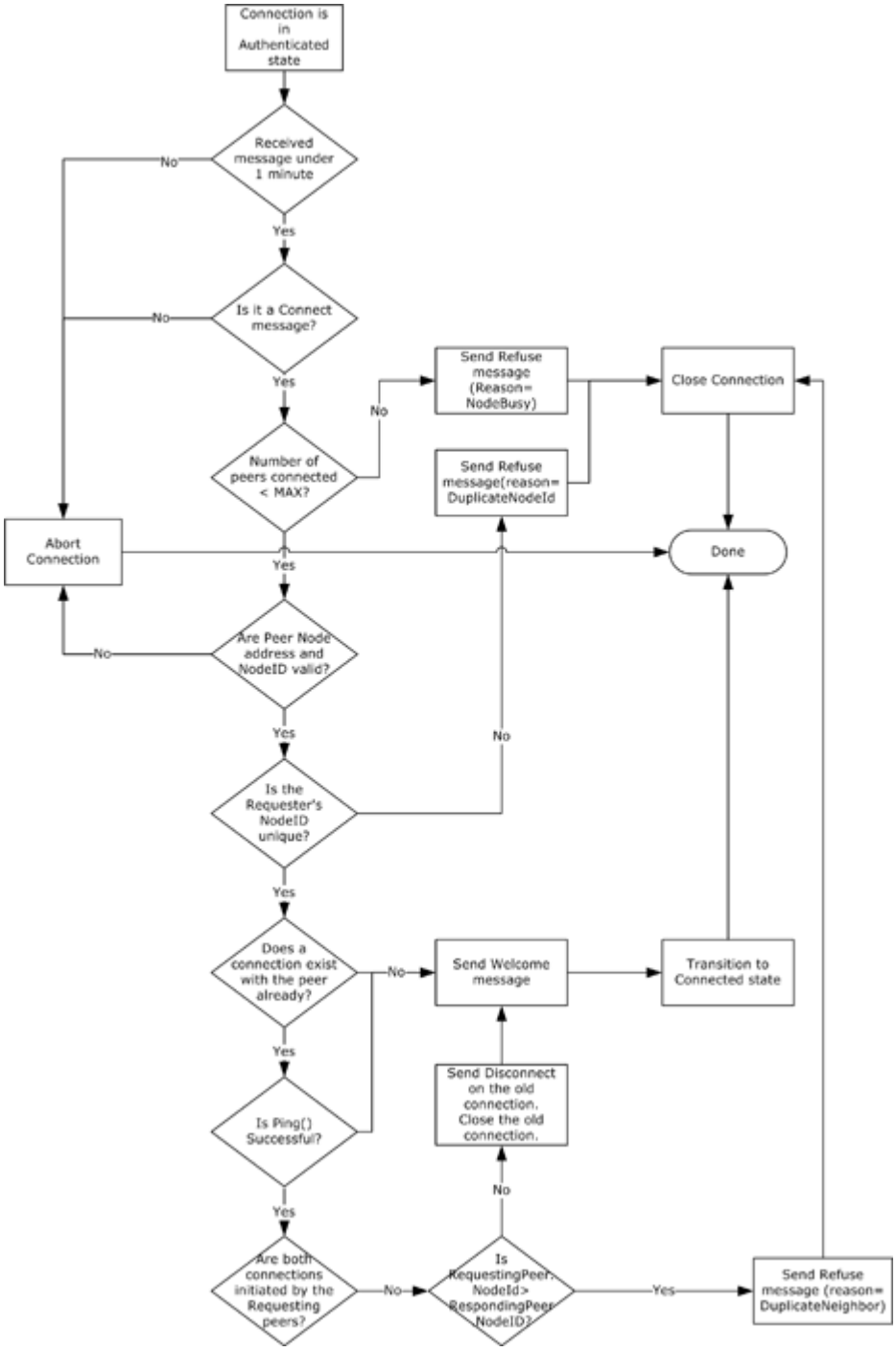

Figure 2: Flow chart of connection process for responding node

3.1.5.4 Receiving a Welcome Message

A Welcome message is sent as a response to a Connect message if the responding node is willing to accept the neighbor connection. A Welcome message MUST be processed as follows by the receiver.

1. If the neighbor connection is not in Authenticated state, or if the receiving node is not the initiator of the connection, the receiving node MUST abort the connection and stop the protocol.

2. If the **NodeId** received in the message is zero, the receiving node MUST send a Disconnect message with the **Reason** element set to "InvalidNeighbor" and close the connection.

3. If the **NodeId** received is the same as the **NodeId** of the receiving node, a Disconnect message with the **Reason** element set to "DuplicateNodeId" MUST be sent and the connection MUST be closed.

4. If a valid neighbor connection to a node with the same **NodeId** (as received in the message) already exists (called old connection), one of the connections MUST be closed. A connection to close MUST be chosen as follows.

    1. If both connections are initiated by the same node, the new connection MUST be picked. In this case, a Refuse message with the **Reason** element set to "DuplicateNeighbor" MUST be sent on the new connection. The connection MUST be closed.

    2. Otherwise, the connection that was initiated by the node whose **NodeId** is greater MUST be picked for closing. A Disconnect message MUST be sent with the **Reason** element set to "DuplicateNeighbor". The neighbor connection MUST be closed.

5. The receiving node MUST validate that the Referral collection is properly formatted. If the referral validation fails, A Refuse message with the **Reason** element set to "InvalidNeighbor" MUST be sent along with the receiving node's Referral set. The neighbor connection MUST be closed, and the receiving node MUST stop processing the message further.

6. The receiving node MUST cache the Referrals from the message.

7. The receiving node MUST change the neighbor's state to Connected.

3.1.5.5 Receiving a Refuse Message

A receiving node MUST process the Refuse message as follows.

1. If the neighbor connection is not in Authenticated state, or the receiving neighbor is not the initiator of the connection, the connection MUST be aborted and the protocol terminated.

2. If the RefuseReason specified in the message is not valid, the Referrals MUST not be used.

3. Otherwise, the receiving node SHOULD cache the Referrals received in the message.

4. The receiving node MUST close the connection.

3.1.5.6 Receiving a Disconnect Message

A receiving node MUST process a Disconnect message as follows.

1. If the neighbor connection is not in the Connected state, the connection MUST be aborted.

2. If the **DisconnectReason** identified in the message is illegal, the connection MUST be aborted.

3. The receiving node SHOULD cache the Referrals received in the message.

4. The receiving node MUST close the connection.

3.1.5.7 Receiving a LinkUtility Message

A receiving node MUST process a LinkUtility message as follows.

1. If the neighbor connection is not in Connected state, the node MUST stop processing the message and abort the neighbor connection.

2. The node MUST validate the incoming message for the counts to be within the bounds. If the message identifies a Total message count that is more than the messages sent by this node, if the useful count is more than total, or if the message identifies a total message count of more than 32, the message MUST be considered as invalid. In this case, the node MUST stop processing the message and abort the connection.

3. The receiving node SHOULD adjust the LinkUtility value<6> of the neighbor connection for both useful and not useful messages.

4. Adjust the Total messages pending acknowledgement to reflect this LinkUtility message.

3.1.5.8 Receiving a Ping Message

A node MUST NOT send any response to the Ping message. Any additional fields contained in the message MUST be ignored. The Ping message is only used to validate that a connection between two neighbors is still valid.

3.1.5.9 Receiving a Flood Message

For each ChannelType instance in the node that has the matching URI (with the PeerVia header value in the message), a copy of the flood message is dispatched for processing. The following steps MUST be taken to process the flood message.

1. The neighbor connection on which the flood message was received MUST be in Connected state. If not, the node MUST drop the message, and the neighbor MUST be aborted.

2. Verify that the message has a valid FloodMessage header. If the header does not exist or is formatted incorrectly, the message MUST be dropped, and the neighbor connection MUST be closed.

3. Verify that the message has a valid PeerVia header and that the value is a valid URI. If the message does not satisfy these checks, the message MUST be dropped, and the neighbor connection MUST be closed.

4. Determine if the message is expected to be signed. This SHOULD be determined based on the URI value in the PeerVia header. If the message is expected to be signed, verify the message signature. If the message is not signed, or if the signature check has failed, the node MUST drop the message and abort the neighbor connection. If the message is determined to have a valid signature, the node MUST use the signature bytes as the unique identifier for the message.

5. If the message is not expected to be signed, read the value in the MessageId header as the unique identifier of the message.

6. Determine if the message is a duplicate using the unique identifier. Consult the MessageId cache<7> to see if a message with the same unique identifier has been processed by the node before. The node SHOULD update the link utility if the previous copy of the message was received more than two minutes ago. If the message is a duplicate, the node SHOULD continue processing at step 10.

7. The node MUST deliver the message to the locally registered endpoint for processing.

8. Examine the message for a HopCount header. If the header is present, its value must be a valid unsigned long. If the header is present and the value is invalid, the node MUST stop processing the message, and MUST abort the neighbor connection.

9. If the Hops header value is greater than one, the node SHOULD forward the message to its neighbors as follows: Remove the HopCount header from the message. Create a HopCount header whose value is exactly one less than the initial value received in the message. Attach the new header to the message. For each neighbor connection (other than the neighbor that sent the flood message) in connected state, the node SHOULD send the message.

10. Update LinkUtility for the neighbor connection that sent the message.

11. Run the LinkUtility protocol, if needed.

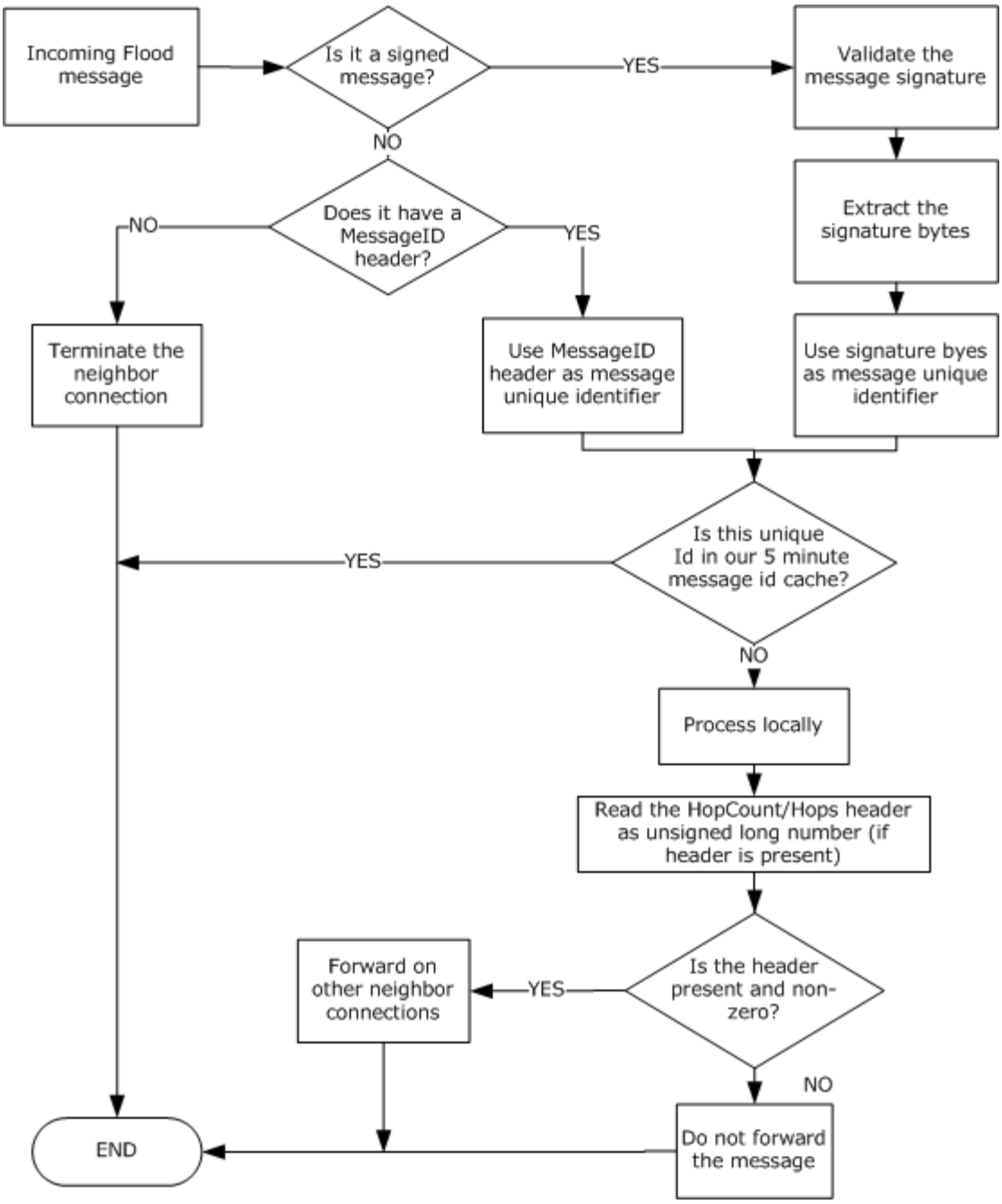

Figure 3: Flow chart of the processing of a flood (application) message

Throttling: Because all unique messages (either originating at the node or received by the node from its node) must be forwarded to its neighbors, there is a possibility of too many buffers being consumed by the pending messages. Such an uncontrolled message flow may lead to the current node's process failure due to low memory caused by busy message buffers. The node MUST use a throttling mechanism to control the amount of memory allocated for message processing.

Messages subject to throttling include those being received and those waiting in the queue to be delivered on the neighbor connections. (A message being sent on all neighbor connections counts as one message). The node MUST set a limit<8> on the number of messages that can be pending at any given time.

When this throttling limit is reached, the node MUST take the following steps to recover from the backlog of messages.

- The node MUST stop receiving messages.
- The node MUST attempt to determine the slowest neighbor, defined as the neighbor that has the most number of pending messages and is in the Connected state. If the slowest neighbor's number of pending messages is low enough, the node MAY<9> choose to cancel throttling, and then resume receiving messages. This approach assumes that the backlog of messages is a result of a transient condition that has already been cleared.
- The node MUST give the slowest neighbor a "grace period" in which the slowest neighbor must improve on its backlog of pending messages. The length of the grace period and the improvement required by the end of it are implementation-specific.<10>

The node MAY<11> also actively monitor the number of pending messages at the slowest neighbor. If the number of pending messages at the slowest neighbor drops below a certain value, the node MAY cancel neighbor monitoring, cancel the grace period, and resume receiving messages.

When the grace period ends, if the slowest neighbor has not satisfied the conditions established in step 3, the neighbor connection MUST be aborted.

The node MUST ensure that the pending message count drops below the preconfigured throttle limit (determined by implementation) before message receiving is resumed.

3.1.6 Timer Events

3.1.6.1 Security Handshake Timer

When a newly established connection's Security Handshake timer expires, the connection MUST be aborted, and the neighbor state information regarding that connection MUST be deleted. Messages received on the connection MUST be dropped.

3.1.6.2 Connect Handshake Timer

When the Connect Handshake timer (that is created for a neighbor connection) expires, the connection MUST be aborted, and the neighbor state information regarding that connection MUST be deleted. Messages received on the connection MUST be dropped.

3.1.6.3 LinkUtility Timer

When the LinkUtility timer (for a particular neighbor connection) expires, the node MUST send a LinkUtility message containing the current LinkUtility<12> value to the neighbor. However, if no messages from that neighbor have been received since the last firing of the LinkUtility timer, the node MUST NOT send a LinkUtility message.

3.1.6.4 Maintenance Timer

When the Maintenance timer expires, the node MUST employ a maintenance algorithm to ensure that it has a useful set of connections to the mesh. The algorithm MUST prefer to remain connected to neighbors that have a higher utility to the mesh as computed through the LinkUtility calculation.

The algorithm MUST ensure that the node has at least IdealNeighborCount neighbors where possible, and no more than MaxNeighborCount.

- IdealNeighborCount SHOULD be 3.
- MaxNeighborCount SHOULD be 7.
- MinNeighborCount SHOULD be 1.

The Maintenance algorithm MUST prune excess connections by sending a Disconnect message. The **Reason** element MUST be set to "NotUsefulNeighbor".

When new connections are required, the node SHOULD<13> discover nodes to connect to by examining a local cache of Referrals returned from previous connection attempts. If Referral Sharing mode is turned OFF or if the local referral cache is empty, it MUST use the discovery service to locate nodes.

3.1.6.4.1 Maintenance Algorithm

The follow procedure SHOULD be used during maintenance to ensure that a node has a set of useful connections to the mesh.

1. If the node has a neighbor count equal to IdealNeighborCount, the node MUST skip to step 4.
2. If the neighbor count is greater than IdealNeighborCount, the node MUST perform the Pruning procedure.
3. If the neighbor count is less than IdealNeighborCount, the node MUST establish new neighbor connections. Connection information in the node's referral cache SHOULD be used first, and then more connection information (if needed) SHOULD be acquired via a discovery service.
4. The node MUST ensure that the maintainer itself is not closing (if the Maintenance timer fires during the closing of the node), and then MUST reset the Maintenance timer. If this is the first time maintenance has been run for this node, the timer SHOULD be set to 10 seconds. Otherwise, the timer SHOULD be set to 5 minutes.

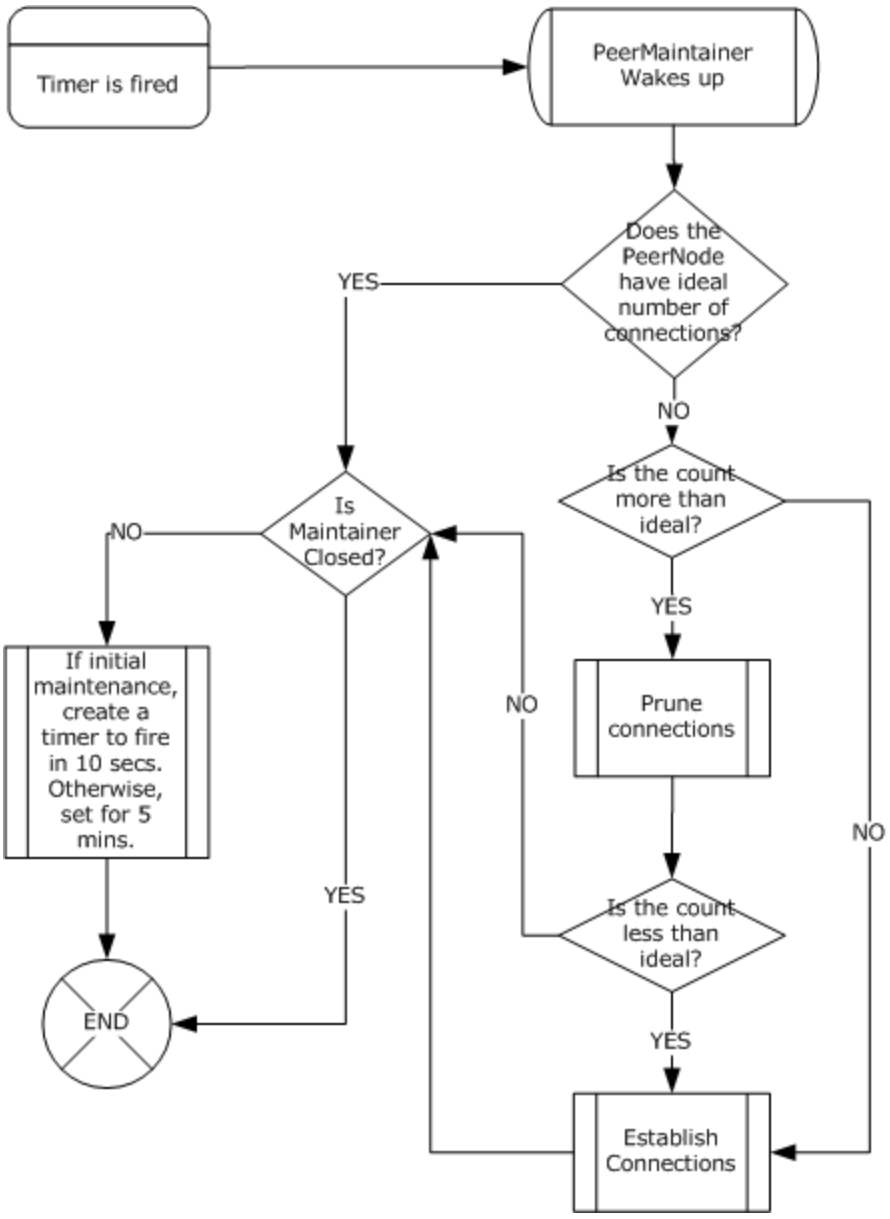

Figure 4: Peermaintainer procedure

3.1.6.4.2 Pruning Algorithm

The following procedure SHOULD be used in the case in which, during maintenance, a node has a neighbor count greater than an IdealNeighborCount.

1. If, at any point, the node begins to close (in case the node tries to shut down during pruning), the node MUST exit the algorithm.

2. If the node has a neighbor count equal to IdealNeighborCount, the node MUST exit the algorithm.

3. Determine the node's least useful neighbor. This MUST be defined as the neighbor with the lowest LinkUtilityIndex who has sent at least 32 messages. If such a neighbor exists, the node MUST close the connection with this neighbor, and then go to step 2. Otherwise, the node MUST exit the algorithm.

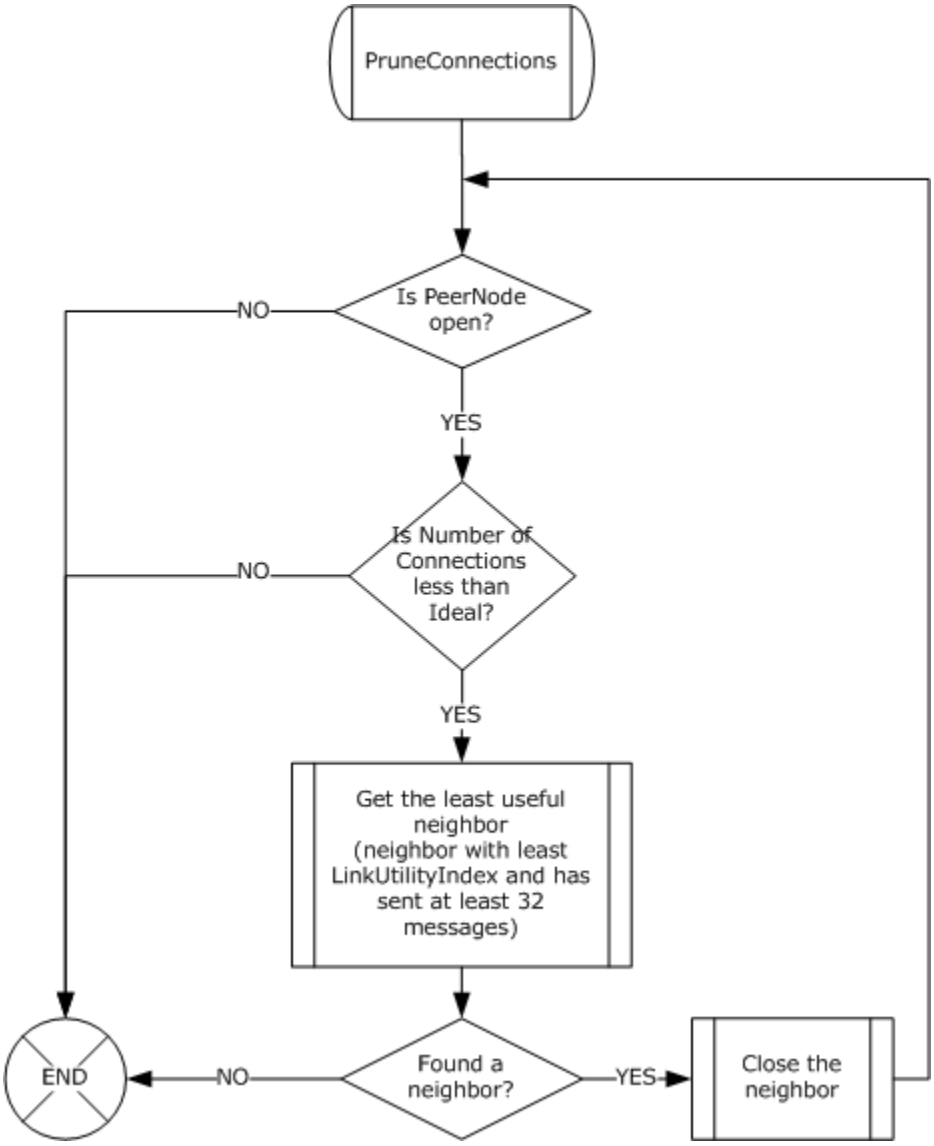

Figure 5: Neighbor pruning procedure

3.1.6.4.3 Establish a Neighbor Connection

The requesting node MUST open a TCP/IP connection with the responding node by doing the following.

The node MUST determine what type of connection to establish:

- No Security

- Password-Based Security
- Certificate-Based Security

After the type of connection has been established, follow the appropriate connection protocol defined in the following sections.

3.1.6.4.4 Create a TCP/IP Connection

To create a TCP/IP connection, follow these steps.

1. Sort the IPAddress list in the PeerNodeAddress for connection reliability in descending order (most reliable first), as specified in [RFC3484].
2. Start the connect timer to expire after one minute.
3. For each IPAddress in the result list, do the following.
    1. If there are no IP addresses, the connection MUST fail.
    2. Create a valid URI by substituting the first IPAddress in the list as the HostName property in the PeerNodeAddress Address field (URI published by the node). The IPAddress MUST be removed from the list.
    3. If the connection type is "No security", attempt to establish a TCP connection to this URI.
    4. Otherwise, attempt to establish a TLS connection, as specified in [RFC4346].
    5. If the connect timer expired, the connection MUST fail.
    6. If the connect attempt failed with a transient error (if the endpoint is not found or the address is unreachable, as specified in [MS-ERREF]), the node SHOULD restart the connection attempt from step (a).
    7. If the connect attempt failed for security reasons (if requesting a TLS over TCP connection and the certificate credentials could not be validated), the connection attempt MUST be failed.
    8. If the connection attempt succeeded, exit for each.

3.1.6.4.5 No Security

The node MUST create a TCP/IP connection to the responding node. The Connect handshake MUST be initiated by the requesting node immediately after the neighbor connection is successfully established.

On successful completion of the Connect handshake, the node MUST be prepared to send and receive all Peer Channel Protocol messages.

3.1.6.4.6 Password-Based Security

The requesting node MUST provide an X509 certificate to secure the connection. The node MUST create a TCP/IP connection to the responding node. The requesting node MUST initiate the Password-Based Security handshake.

On successful completion of the Password-Based Security handshake, the requesting node MUST initiate the Connect handshake.

On successful completion of the Connect handshake, the node MUST be prepared to send and receive all Peer Channel Protocol messages.

The following diagram identifies the state transitions of a neighbor-to-neighbor connection in Password-Based Security mode.

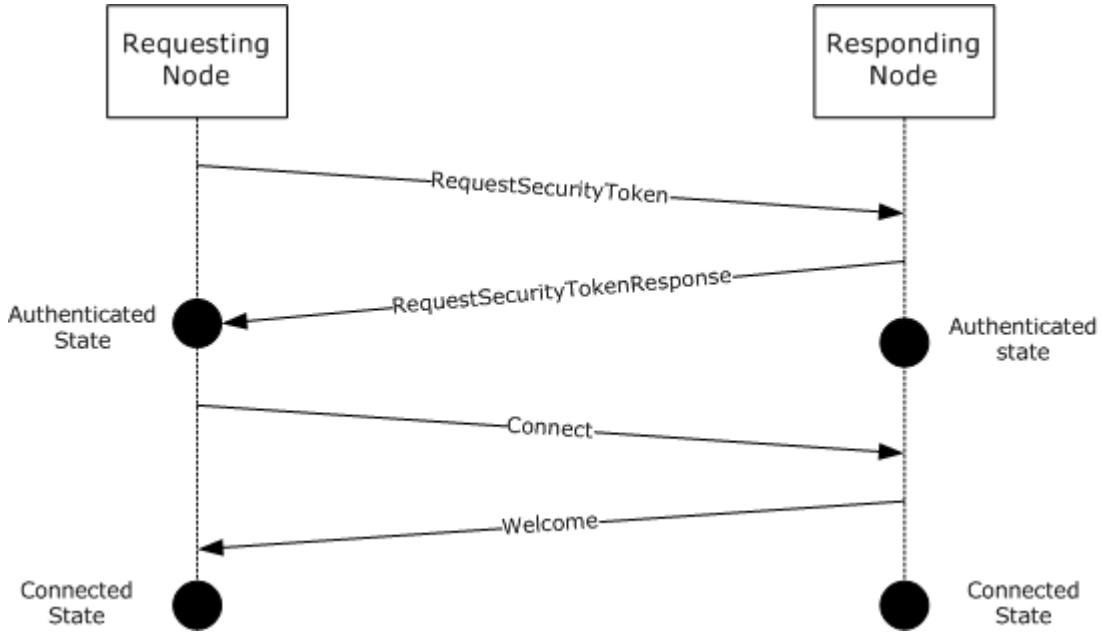

Figure 6: Neighbor connection handshake using password-based security

3.1.6.4.7 Certificate-Based Security

The higher-level application or protocol MUST provide an X509 certificate to secure the connection. The node MUST create a TCP/IP connection to the responding node. The requesting node MUST initiate the Connect handshake. On successful completion of the Connect handshake, the node MUST be prepared to send and receive all Peer Channel Protocol messages.

3.1.6.4.7.1 Password-Based Security Handshake

After creating a connection (TLS over TCP with anonymous X509 certificates), the requesting node MUST send a RequestSecurityToken to prove the knowledge of the password. The responding node MUST respond to this message by replying with a RequestSecurityTokenResponse message.

At this point, both nodes MUST transition this connection to the Authenticated state. A Connect handshake MUST then be initiated by the requesting node. A Connect Handshake timer MUST be started.

3.1.6.4.7.2 Connect Handshake

The requesting node's connection goes through the following transitions once the connection is in Authenticated state.

- The requesting node MUST start a Connect Handshake timer.

- The requesting node MUST send a Connect message within one minute of establishing the connection.

The Address field of the Connect message MUST be set to the PeerNodeAddress constructed when the node was opened (see section 3.1.4.1).

The NodeId field of the Connect message must be set to a unique NodeId to identify this node.

- The requesting node must wait for up to 1 minute for a response. It SHOULD NOT wait for more than one minute. If a response is not received within this time frame, the requesting node SHOULD abort the connection.
- The responding node MUST respond with either a Welcome message or a Refuse message. If the requestor receives a Welcome message, the connection transitions to the Connected state. Both nodes can now exchange Peer Channel Protocol messages. If a Refuse message is received, the requesting node MUST close the connection.
- If the responding node included Referrals in the return message, they SHOULD be added to the Referrals cache to aid in establishing additional neighbor connections in the future.

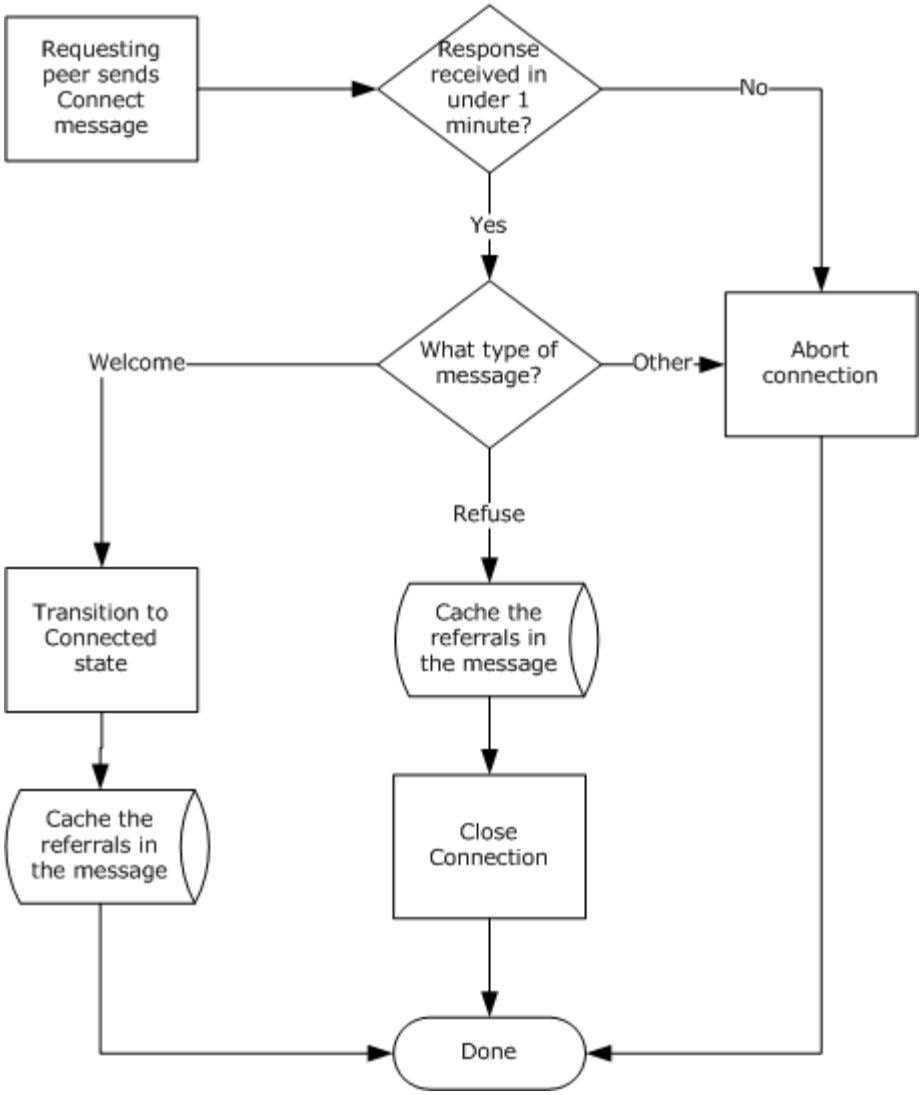

Figure 7: Flow chart of requesting node's connection process

The following diagram shows the state transitions of a node through the connection process.

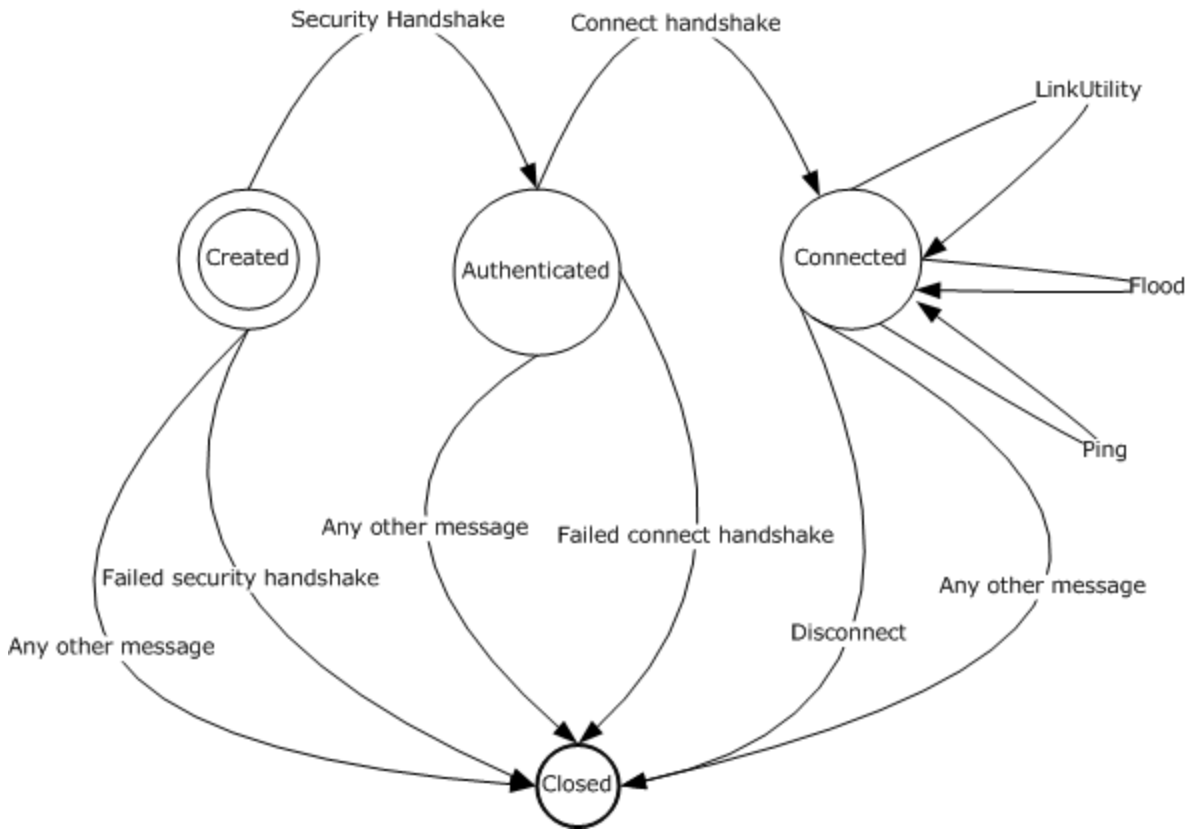

Figure 8: State transitions of node during connection process

3.1.7 Other Local Events

There are no other local events to be defined for this protocol.

3.2 Sender Details

The sender role is a superset of the message processing and data model. Senders follow all the message processing rules of receivers that are defined in the previous section. In addition, senders MUST be able to send flood messages to the mesh. This is triggered by a higher-layer initiated action. Any sender-related specification here is in relation to the sender's role that is a superset of the receiver's role (see section 3.1).

3.2.1 Abstract Data Model

The sender abstract data model is the same as the receiver abstract data model (see section 3.1.1).

3.2.2 Timers

Same as receiver timers.

3.2.3 Initialization

For receiver initialization, see section 3.1.3.

3.2.4 Higher-Layer Triggered Events

The sender has one additional higher-layer triggered event, which is the sending of an application message.

3.2.4.1 Sending Messages

Flood messages are exchanged between nodes as a result of application generating messages to be sent to the mesh. When a higher layer passes a message to the node, it adds the following headers to the application message.

Name	Purpose	Value
MessageID	To carry a GUID that uniquely identifies the message in the mesh.	Each application message MUST contain this header with a unique GUID that identifies the message.
FloodMessage	To identify that this is an application message.	This header must contain the value "PeerFlooder" as the ONLY text node in its element.
PeerVia	To identify the target ChannelType of the message.	This contains the intended target site of the message. MUST contain the URI of the application listening endpoint. Typically, this is the value of the Via property before Peer Channel processed the outgoing message.
PeerTo	Application-specific target for the message.	This SHOULD be set to the same value as PeerVia.

The Peer Channel Protocol allows multiple ChannelType registrations on the same node that participate in the same mesh. This means that a single Peer Channel Protocol endpoint may act as a multiplexer and send messages destined for different ChannelTypes.

3.2.4.1.1 Sending Signed Messages

After adding the flood headers, the application message (excluding the PeerHopCount header) MUST secure the message (for more information, see [MSDN-SECURITY_INFORMATION]). Receiving nodes use the signature bytes as the unique identifier of the message.

A node sends a LinkUtility message when it has received 32 flood messages on the connection. When 32 messages are received, the LinkUtility message is sent to the remote node, and the LinkUtilityInfo is reset. If a minute has elapsed before receiving 32 messages, and at least one message has been received on that connection during that minute, a LinkUtility message is sent with the current values in the LinkUtilityInfo, and the LinkUtilityInfo is reset.

After performing the above transformations on the message, the node sends the message to its immediate neighbors. Those neighbors, in turn, send the message to their neighbors, and so on. In this manner, the message is **flooded** through the mesh. The flood comes to an end when all nodes that received the message do not forward it any further.

3.2.5 Message Processing Events and Sequencing Rules

See section 3.1.5.

3.2.6 Timer Events

There are no timer events to be defined for this protocol.

3.2.7 Other Local Events

There are no other local events to be defined for this protocol.

4 Protocol Examples

4.1 Establishing a Neighbor Connection in Password Mode

When the mesh is password secured, first the Password-Based Security handshake takes place. After a successful security handshake, the Connect handshake follows.

4.1.1 Connection Initiator Sends the RequestSecurityToken Message

An example of a RequestSecurityToken message follows. It gives the layout of a Request Security token.

```
(00) <s:Envelope xmlns:a="http://www.w3.org/2005/08/addressing" xmlns:
s="http://www.w3.org/2003/05/soap-envelope">
(01) <s:Header>
(02) <a:Action s:mustUnderstand="1">RequestSecurityToken</a:Action>
(03) <a:MessageID>urn:uuid: b3d053cc-eced-43ee-acc1-6c836e219f36 </a:M
essageID>
(04) <a:ReplyTo>
(05) <a:Address> http://www.w3.org/2005/08/addressing/anonymous</a:Add
ress>
(06)    </a:ReplyTo>
(07)  </s:Header>
(08)  <s:Body>
(09) <t:RequestSecurityToken  xmlns:t="http://schemas.xmlsoap.org/ws/2
005/02/trust">
(10) <t:TokenType> http://schemas.microsoft.com/net/2006/05/peer/peerh
ashtoken</t:TokenType>
(11) <t:RequestType> http://schemas.xmlsoap.org/ws/2005/02/trust/Valid
ate</t:RequestType>
(12) <t:RequestedSecurityToken>
(13) <peer:PeerHashToken xmlns:peer="http://schemas.microsoft.com/net/
2006/05/peer">
(14) <peer:Authenticator> mZyMZdkfrFWX+EMp4Lp+eX3sy61391MA15Iqx/9U7yQ=
</peer:Authenticator>
        </peer:PeerHashToken>
      </t:RequestedSecurityToken>
    </t:RequestSecurityToken>
  </s:Body>
</s:Envelope>
```

The following notes give more detail on interesting elements of this message.

00 – Start of the SOAP message.

01 – Start of the header section.

02 – Action value for a RequestSecurityToken message.

08 – Beginning of the SOAP message body.

09 - From here on, the rest of the message shows the formatting of PeerHashToken in a RequestSecurityToken message. This line indicates the beginning of the RequestSecurityToken element.

10 – Indicates the token type of the message. Only "http://schemas.microsoft.com/net/2006/05/peer/peerhashtoken" is allowed for the Peer Channel Protocol RequestSecurityToken message.

11 – Indicates the RequestType. Of all of the valid RequestType options, only "http://schemas.xmlsoap.org/ws/2005/02/trust/Validate" is allowed.

12 – Start of the RequestSecurityToken message. Acts as the parent element for the type of token being carried in this message. The Peer Channel Protocol only allows "http://schemas.microsoft.com/net/2006/05/peer/PeerHashToken: elements.

13 – Demonstrates the PeerHashToken element.

14 – PeerHashToken carries an Authenticator element that carries the HMAC value computed based on the public key and the hash of the password (see next section).

4.1.2 Responding Node Sends Back RequestSecurityTokenResponse

An example of a RequestSecurityTokenResponse message follows.

```
(00) <s:Envelope xmlns:s="http://www.w3.org/2003/05/soap-enve
lope" xmlns:a="http://www.w3.org/2005/08/addressing">
(01)  <s:Header>
(02)    <a:Action s:mustUnderstand="1">RequestSecurityTokenResponse</a
:Action>
(03)    <a:RelatesTo>urn:uuid:b3d053cc-eced-43ee-acc1-6c836e219f36</a:
RelatesTo>
(04)    <a:To s:mustUnderstand="1">http://www.w3.org/2005/08/addressi
ng/anonymous</a:To>
(05)  </s:Header>
(06)  <s:Body>
(07)    <t:RequestSecurityTokenResponse xmlns:t="http://schemas.xmlsoa
p.org/ws/2005/02/trust" xmlns:u="http://docs.oasis-open.org/wss/2004/0
1/oasis-200401-wss-wssecurity-utility-1.0.xsd">
(08) <t:TokenType> http://schemas.microsoft.com/net/2006/05/peer/peerh
ashtoken</t:TokenType>
(09)      <t:Status>
(10)<t:Code> http://schemas.xmlsoap.org/ws/2005/02/trust/status/valid<
/t:Code>
(11)      </t:Status>
(12)      <t:RequestedSecurityToken>
(13) <peer:PeerHashToken xmlns:peer="http://schemas.microsoft.com/net/
2006/05/peer">
(14) <peer:Authenticator> Z9Mbuum3+S/uoCrG2611nIvHiKC9Aj/NCmqscaOoQao=
</peer:Authenticator>
        </peer:PeerHashToken>
      </t:RequestedSecurityToken>
    </t:RequestSecurityTokenResponse>
  </s:Body>
</s:Envelope>
```

The following notes give more detail on interesting elements of this message.

02 – Action header. Must be set to "RequestSecurityTokenResponse".

03 – RelatesTo header identifying the MessageID of the corresponding RequestSecurityToken message (see previous section).

07 – RequestSecurityTokenResponse element. Start of the body containing the response.

08 – Identifies the token type. Must be the same token type as what is in the RequestSecurityToken message.

09 – Start of the Status element. This element contains the result of the validation of the requesting node's token.

10 – Start of the "Code" element. Indicates the status code. Note that the only legal value is "http://schemas.xmlsoap.org/ws/2005/02/trust/status/valid". In error cases, a reply message must not be sent by the responding node. Instead, the responding node must close the connection.

12 – Start of the "RequestedSecurityToken" element. This contains the response of the responding node. This must contain the PeerHashToken of the responding node. The hash that the requesting node separately computes for the responding party must match this value for the security handshake to succeed.

13 – Start of the PeerHashToken element.

14 – Authenticator element containing the hash.

4.1.3 Requesting Node Sends a Connect Message

Now that the Password-Based Security handshake is successful, the requesting node sends a Connect message.

An example of a Connect message follows.

```
<s:Envelope xmlns:s="http://www.w3.org/2003/05/soap-envelope" xmlns:a=
"http://www.w3.org/2005/08/addressing">
  <s:Header>
    <a:Action s:mustUnderstand="1">http://schemas.microsoft.com/net/20
06/05/peer/Connect</a:Action>
    <a:To s:mustUnderstand="1">net.p2p://securechatmesh/</a:To>
  </s:Header>
  <s:Body>
    <Connect xmlns="http://schemas.microsoft.com/net/2006/05/peer" xml
ns:i="http://www.w3.org/2001/XMLSchema-instance">
      <Address>
        <EndpointAddress>
          <a:Address>net.tcp://160.20.30.40:63758/Peer ChannelEndpoint
s/ba703e02-6a7b-457c-bf81-f0d6e56adb97</a:Address>
        </EndpointAddress>
        <IPAddresses xmlns:b="http://schemas.datacontract.org/2004/07/
System.Net">
          <b:IPAddress>
            <b:m_Address>0</b:m_Address>
            <b:m_Family>InterNetworkV6</b:m_Family>
            <b:m_HashCode>0</b:m_HashCode>
            <b:m_Numbers xmlns:c="http://schemas.microsoft.com/2003/10
/Serialization/Arrays">
              <c:unsignedShort>8193</c:unsignedShort>
              <c:unsignedShort>125</c:unsignedShort>
```

```
              <c:unsignedShort>40</c:unsignedShort>
              <c:unsignedShort>3</c:unsignedShort>
              <c:unsignedShort>246</c:unsignedShort>
              <c:unsignedShort>345</c:unsignedShort>
              <c:unsignedShort>456</c:unsignedShort>
              <c:unsignedShort>567</c:unsignedShort>
            </b:m_Numbers>
            <b:m_ScopeId>0</b:m_ScopeId>
          </b:IPAddress>
          <b:IPAddress>
            <b:m_Address>3750312861</b:m_Address>
            <b:m_Family>InterNetwork</b:m_Family>
            <b:m_HashCode>0</b:m_HashCode>
            <b:m_Numbers xmlns:c="http://schemas.microsoft.com/2003/10
/Serialization/Arrays">
              <c:unsignedShort>0</c:unsignedShort>
              <c:unsignedShort>0</c:unsignedShort>
              <c:unsignedShort>0</c:unsignedShort>
              <c:unsignedShort>0</c:unsignedShort>
              <c:unsignedShort>0</c:unsignedShort>
              <c:unsignedShort>0</c:unsignedShort>
              <c:unsignedShort>0</c:unsignedShort>
              <c:unsignedShort>0</c:unsignedShort>
            </b:m_Numbers>
            <b:m_ScopeId>0</b:m_ScopeId>
          </b:IPAddress>
          <b:IPAddress>
            <b:m_Address>0</b:m_Address>
            <b:m_Family>InterNetworkV6</b:m_Family>
            <b:m_HashCode>0</b:m_HashCode>
            <b:m_Numbers xmlns:c="http://schemas.microsoft.com/2003/10
/Serialization/Arrays">
              <c:unsignedShort>65152</c:unsignedShort>
              <c:unsignedShort>0</c:unsignedShort>
              <c:unsignedShort>0</c:unsignedShort>
              <c:unsignedShort>0</c:unsignedShort>
              <c:unsignedShort>27023</c:unsignedShort>
              <c:unsignedShort>28969</c:unsignedShort>
              <c:unsignedShort>48266</c:unsignedShort>
              <c:unsignedShort>44428</c:unsignedShort>
            </b:m_Numbers>
            <b:m_ScopeId>0</b:m_ScopeId>
          </b:IPAddress>
        </IPAddresses>
      </Address>
      <NodeId>14800704070183415334</NodeId>
    </Connect>
  </s:Body>
</s:Envelope>
```

4.1.4 Responding Node Sends a Welcome Message

The responding node accepts the request and sends back a Welcome message.

An example of a Welcome message follows.

```
<s:Envelope xmlns:s="http://www.w3.org/2003/05/soap-envelope" xmlns:a=
"http://www.w3.org/2005/08/addressing">
  <s:Header>
    <a:Action s:mustUnderstand="1">http://schemas.microsoft.com/net/20
06/05/peer/Welcome</a:Action>
    <a:To s:mustUnderstand="1">http://www.w3.org/2005/08/addressing/an
onymous</a:To>
  </s:Header>
  <s:Body>
    <Welcome xmlns="http://schemas.microsoft.com/net/2006/05/peer" xml
ns:i="http://www.w3.org/2001/XMLSchema-instance">
      <NodeId>16299239282823246037</NodeId>
      <Referrals></Referrals>
    </Welcome>
  </s:Body>
</s:Envelope>
```

4.2 Non-Password Security Modes

If the mesh is configured with any other transport security mode, the Connect handshake (see section 3.1.6.2) will be the first sequence of messages to be exchanged on the connection.

4.3 Flooding a Message

A sample flood message follows.

```
(00)<s:Envelope xmlns:s="http://www.w3.org/2003/05/soap-envelope" xmln
s:a="http://www.w3.org/2005/08/addressing">
(01) <s:Header>
(02) <a:Action s:mustUnderstand="1">http://MyPeerApplication/MyMethod<
/a:Action>
(03) <a:To s:mustUnderstand="1">net.p2p:// MyPeerApplication/</a:To>
(04) <MessageID xmlns="http://schemas.microsoft.com/net/2006/05/peer">
urn:uuid:271dddd4-fa44-46e2-9b86-090c0a52326c</MessageID>
(05) <PeerTo xmlns="http://schemas.microsoft.com/net/2006/05/peer">net
.p2p://ApplicationMeshName/Channel1/</PeerTo>
(06)   <PeerVia xmlns="http://schemas.microsoft.com/net/2006/05/peer">
net.p2p://ApplicationMeshName/Channel1</PeerVia>
(07)    <FloodMessage xmlns="http://schemas.microsoft.com/net/2006/05/
peer">PeerFlooder</FloodMessage>
(08)   </s:Header>
(09)  <s:Body>
(10)  </s:Body>
(11)</s:Envelope>
```

Other than the following Peer Channel Protocol-specific header (that a node adds to the message; see section 3.2), this message is the same as the application message.

04 - Demonstrates a MessageID header containing a serialized GUID as its element text.

05 - Demonstrates a PeerTo header containing the application target endpoint. On a received message, this header is used to route the message to the appropriate message processing endpoint. It contains the URI of the endpoint that ultimately processes the message.

06 - Demonstrates the PeerVia header containing the URI of the ChannelType that is sending the message. PeerVia and PeerTo must be the same.

07 - Demonstrates the FloodMessage header. This header is always present with "PeerFlooder" as the single text node value, if the message is to be treated as a flood message by the node.

5 Security

The following security modes are available to use with the Peer Channel Protocol.

Transport Security: This mode dictates that the neighbor-to-neighbor connections must be secured with a TLS over TCP connection. There are two modes of transport security.

- X509 Certificate-Based: In this mode, each node will have an X509 certificate that is issued by a well-known authority. The neighbor-transport connection in this case is a TLS connection configured with the above certificate. This certificate is used by the remote party to authenticate the requesting node before allowing the requesting node to join the mesh. Similarly, the requesting node must also authenticate the accepting node (using the certificate that the accepting node has presented during the TLS connection negotiation). For how the Microsoft implementation of the Peer Channel Protocol uses X509 certificates, see appendix A.
- Password-Based: In case the mesh is secured by a password, the transport is still established using TLS over TCP. In this case, any X509 certificate may be used. The nodes must not authenticate each other's certificate. Instead, they must prove knowledge of the password to each other using the password security protocol. The requesting neighbor must initiate the password security protocol as soon as the connection is established.

X509 Certificate-Based Message-Level Security: Independent of the transport security, a mesh can also be configured to have message-level security. In this mode, all senders include a digital signature along with the message. The signature is computed using a well-known X509 certificate credential. The signature is computed over the application message and sent along with the application message. The message is secured, as specified in [WSTrust].

5.1 Security Considerations for Implementers

In a mesh configured with no security, neighbor connections are not authenticated. Also, individual application messages are not signed, which exposes the mesh to tampering attacks.

If a mesh is configured with transport security with password credential, make sure to validate the PeerHashToken using the remote node's public key. This is important to avoid man-in-the-middle type of attacks against the Peer Channel Protocol security handshake.

If a mesh is configured with transport security with trusted X509 certificates, make sure to authenticate the remote node during the connection establishment phase. Any connection being requested by a node with an untrusted certificate must be rejected.

If a mesh is configured with a message integrity check, it is important to verify that the message is not signed. Also, authenticate the credential that is used to sign the message. Typically, this certificate must be trusted by all nodes and must have access to the public of such a certificate.

5.2 Index of Security Parameters

The following security parameters are associated with this protocol:

Security Parameter	Section
PeerHashToken	2.2.1.1
Security Configuration	3.1.1

Security Parameter	Section
Message Identifier	

6 Appendix A: Windows Behavior

The information in this specification is applicable to the following versions of Windows:

- Windows XP
- Windows Server 2003
- Windows Vista

Exceptions, if any, are noted below. Unless otherwise specified, any statement of optional behavior in this specification prescribed using the terms SHOULD or SHOULD NOT implies Windows behavior in accordance with the SHOULD or SHOULD NOT prescription. Unless otherwise specified, the term MAY implies that Windows does not follow the prescription.

<1> Section 2.2.1.2: Windows writes an arbitrary value in the IPAddress/m_HashCode element when serializing a PeerNodeAddress instance. On deserializing a PeerNodeAddress, Windows ignores the value in the IPAddress/m_HashCode element.

<2> Section 3.1.2: When a node is first opened, Windows performs Initial maintenance. Initial maintenance attempts to establish two neighbor connections. If it fails to establish any neighbor connectionsWindows schedules Periodic maintenance to run after ten seconds, otherwise Windows schedules Periodic maintenance to run after five minutes.

<3> Section 3.1.2: Windows attempts to ensure a well connected mesh by performing a Periodic maintenance routine every five minutes.

<4> Section 3.1.2: If the number of connected neighbors falls to zero, Windows performs Periodic maintenance immediately.

<5> Section 3.1.3.1: PeerNode selects addresses available on the local machine to publish as referrals and with a discovery service using the following algorithm.

1. Do not select addresses that are configured as:
   - Loopback addresses
   - Transient addresses
   - Having a random suffix origin
2. If the remaining address set is non-empty, reorder them to put at least one of each of the following categories at the top of the list: 6-to-4 address, Teredo address, ISATAP address, ipv6 address, ipv4 address.
3. If the remaining address set is empty, return addresses that are Transient and have a random suffix origin.

<6> Section 3.1.5.7: The Windows implementation uses the following algorithm to calculate the LinkUtility value of a neighbor connection.

1. LinkUtility of a connection is initialized to zero.
2. For each transmitted or received message, the following calculation is performed.

```
Un = (Un * 31) / 32 + (useful * 128)
```

```
Where:

    Un  = LinkUtilityIndex.

    Useful = {One, if the message
      was a useful message; otherwise, zero.}
```

<7> Section 3.1.5.9: Windows implementations use node maintains a cache of message IDs of previously processed messages organized as 5 hash tables. At the beginning of each minute, the next table is picked from the cache to be the current table. The content of the table is cleared (that is, all messages received five minutes ago are forgotten). An incoming message's ID is checked in all tables for a match. If there is a match in any of the tables (that is, if a message with the same ID is seen within the last five minutes), the message is deemed duplicate. An incoming non-duplicate flood message's ID is added to the current table.

<8> Section 3.1.5.9: The Windows implementation of throttling initiates if more than 128 messages are pending at the local node.

<9> Section 3.1.5.9: The Windows implementation of throttling is cancelled if the slowest neighbor has 32 or less pending messages.

<10> Section 3.1.5.9: The Windows implementation of message throttling gives the slowest neighbor a grace period of 10-20 seconds (determined randomly) to clear out half of its pending messages.

<11> Section 3.1.5.9: The Windows implementation of message throttling actively monitors the number of pending messages at the slowest neighbor. If the number of pending messages drops to 8 or below at any point during the grace period, neighbor monitoring is discontinued, the grace period timer is cancelled, and message reception at the local node resumes.

<12> Section 3.1.6.3: The Windows implementation uses the following algorithm to calculate the LinkUtility value of a neighbor connection:

LinkUtility of a connection is initialized to zero.

For each transmitted or received message, the following calculation is performed.

```
Un = (Un * 31) / 32 + (useful * 128)

Where:

Un  = LinkUtilityIndex.

 Useful = {One, if the
   message was a useful message;
otherwise, zero.}
```

<13> Section 3.1.6.4: Windows has a maximum Referral cache size of 50 neighbors.

7 Index

V

W