Wireless Access and Terminal Mobility in CORBA

October 2004
Version 1.2
dtc/04-09-02

Note – This convenience report only contains Chapters 1 and 7 that are different from the Version 1.1 (formal/04-04-02)

Overview 1

This document specifies an architecture and interfaces to support wireless access and terminal mobility in CORBA.

Contents

This chapter contains the following sections.

Section Title	Page
"Design Rationale"	1-1
"Proof of Concept"	1-2
"References"	1-2

1.1 Design Rationale

The basic design principles have been client-side ORB transparency and simplicity. Transparency of the mobility mechanism to non-mobile ORBs has been the primary design constraint. We have rejected all solutions that would require modifications to a non-mobile ORB in order for it to interoperate with CORBA objects and clients running on a mobile terminal. In other words, a stationary (non-mobile, or fixed network) ORB does not have to implement this specification in order to interoperate with CORBA objects and clients running on mobile terminals.

The specification was designed to provide a minimal useful functionality for CORBA applications, in which the client, the server, or both of them are running on a host that can move.

1.2 Proof of Concept

The design is heavily affected by experiences of the EC/ACTS project DOLMEN (AC036) that implemented a prototype of CORBA extensions to support terminal mobility. The DOLMEN solution is described, for example, in the OMG Document telecom/98-08-08.

This specification has been implemented by University of Helsinki as an Open Source extension to the MICO Open Source ORB, called MIWCO [MIWCO].

The GIOP over Bluetooth Tunneling Specification has been implemented in the EC/ITEA project Vivian [VIVAN] as an extension to MIWCO.

1.3 References

[BT-SIG] Bluetooth SIG, Specification of the Bluetooth System - Version 1.1, Volume 1 & 2. February 2001, Available: http://www.bluetooth.com/developer/specification/specification.asp.

[GFD] WAP Forum. WAP General Formats Document. WAP Forum document WAP-188-WAPGenFormats, Version 15-Aug-2000.

[MIWCO] MIWCO - An Open Source Implementation of Wireless CORBA. Available: http://www.cs.helsinki.fi/u/kraatika/wCORBA.html.

[RFC 2988] Computing TCP's Retransmission Timer, IETF, RFC 2988, November 2000.

[VIVIAN] VIVIAN Consortium, GIOP Tunneling over Bluetooth L2CAP. Available: http://www-nrc.nokia.com/Vivian/Public/Html/ltp.html.

[WDP] WAP Forum. Wireless Datagram Protocol Specification. WAP Forum Document WAP-201-WDP, Approved Version 19-February-2000.

GIOP Tunneling 7

Contents

This chapter contains the following sections.

Section Title	Page
"Tunnel Establishment"	7-2
"GIOP Tunneling Protocol"	7-2
"TCP Tunneling"	7-20
"UDP Tunneling"	7-20
"WAP Tunneling"	7-25

A GIOP tunnel is the means to transmit GIOP and tunnel control messages between a Terminal Bridge and an Access Bridge. There is only one GIOP tunnel between a given Terminal Bridge and Access Bridge. However, a graceful handoff behavior is defined so that the Terminal Bridge can seamlessly transfer the GIOP Tunnel from the current Access Bridge to a new one. If the terminal can have simultaneous transport connectivity to two Access Bridges, then the Terminal Bridge creates a new tunnel to a new Access Bridge before shutting down the tunnel to the previous Access Bridge.

A tunnel is shared by all GIOP connections to and from the terminal it is associated with. The tunneling protocol allows multiplexing between the GIOP connections.

The GIOP Tunneling Protocol (GTP) is an abstract, transport-independent protocol. It defines message formats for establishing, releasing, and re-establishing (recovery) the tunnel as well as for transmitting and forwarding GIOP messages. The GTP protocol also defines messages for establishing and releasing GIOP connections through the Access Bridge. Figure 7-1 depicts the protocol architecture.

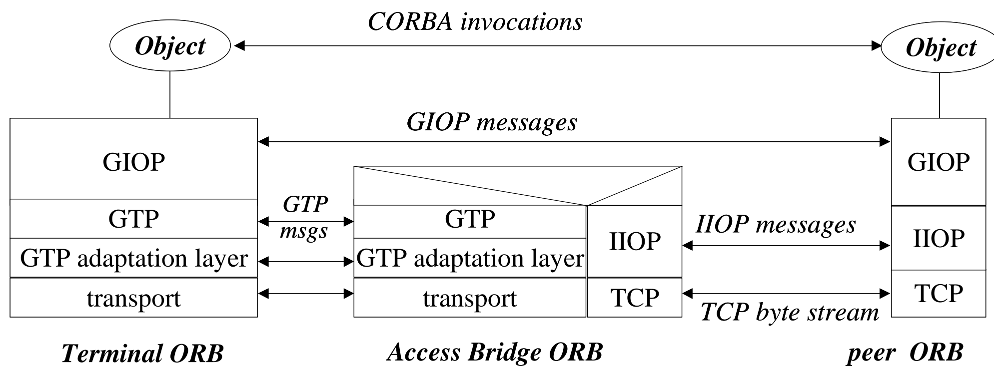

Figure 7-1 GIOP Tunneling Protocol Architecture

Since the GIOP Tunneling Protocol is an abstract protocol, it needs to be mapped onto one or more concrete protocols. This specification defines four concrete tunneling protocols: TCP Tunneling, UDP Tunneling, WAP Tunneling, and Bluetooth Tunneling.

The GTP is designed so that the specification of a concrete tunneling protocol is simple. The specification of a concrete tunneling protocol is provided as an adaption layer between the GIOP Tunneling Protocol and a transport layer protocol. The adaptation layer needs only to define how the transport is to be used and the data format of the transport address of the transport end-point.

7.1 Tunnel Establishment

GIOP tunnel establishment consists of two phases: 1) Transport end-point detection and 2) Establishment of the GIOP tunnel. Transport end-point detection is discussed below. The establishment of the GIOP tunnel is specified in Section 7.2, “GIOP Tunneling Protocol,” on page 7-2.

7.1.1 Transport End-Point Detection

The detection of transport end-points on the link, network, and transport layers. It also depends on the provider of the Access Bridge. Therefore, transport end-point detection is out of the scope of this specification.

7.2 GIOP Tunneling Protocol

The GIOP Tunneling Protocol (GTP) assumes that the underlying concrete tunneling protocol (that is, the adaption layer between the GTP and a transport protocol) provides the same reliability and ordered delivery of messages assumed by the GIOP. If the underlying transport protocol does not provide this level of service, then the adaption layer that resides between the GTP and the actual transport protocol will provide this level of service.

The version of the GIOP Tunneling Protocol defined in this specification is 1.0 (major 1, minor 0).

All timeout values are in seconds.

7.2.1 GTP Message Structure

All GTP messages contain a header of eight octets and contents of variable (possibly null) length.

The GTP header has the structure of

```
struct GTPHeader {
    octet gtp_msg_type;
    octet flags;
    unsigned short seq_no;
    unsigned short last_seq_no_received;
    unsigned short content_length;
};
```

The **gtp_msg_type** element indicates the GIOP Tunneling Protocol message type. It defines how the receiver should interpret the body of the GTP message.

The flags element indicates the Endianness used in the GTP header and in GTP control messages. The leftmost bit tells the Endianness: 0x00 Big-Endian and 0x80 Little-Endian. The remaining seven bits are reserved for future usage.

The **seq_no** element runs from 1 (0x0001) to 65535 (0xFFFF). The value 0x0000 can only appear in tunnel establishment request messages and an associated reply. The sequence number counting follows the usual modulo arithmetic with the exception that the **seq_no** 0x0001 follows the **seq_no** 0xFFFF.

The **last_seq_no_received** element indicates the highest sequence number of GTP messages received or, in certain cases, processed by the sender.

The **content_length** element (unsigned short) tells the length of the GTP message.

7.2.2 GTP Messages

The GTP Messages are listed in the table below. Descriptions of the messages are given in the subsections that follow.

Table 7-1 GTP Messages

Message name	gtp_msg_type	GTP Level
IdleSync	0x00	1, 2
EstablishTunnelRequest	0x01	1, 2
EstablishTunnelReply	0x02	1, 2
ReleaseTunnelRequest	0x03	1, 2

Table 7-1 GTP Messages

Message name	gtp_msg_type	GTP Level
ReleaseTunnelReply	0x04	1, 2
HandoffTunnelRequest	0x05	2
HandoffTunnelReplyCompleted	0x06	2
OpenConnectionRequest	0x07	1, 2
OpenConnectionReply	0x08	1, 2
CloseConnectionRequest	0x09	1, 2
CloseConnectionReply	0x0A	1, 2
ConnectionCloseIndication	0x0B	1, 2
GIOPData	0x0C	1, 2
GIOPDataError	0x0D	1, 2
GTPForward	0x0E	2
GTPForwardReply	0x0F	2
Error	0xFF	1, 2

7.2.3 IdleSync Message

The **IdleSync** message does not have a message body.

Source

Terminal Bridge and Access Bridge

Description

It is used by the Terminal Bridge and the Access Bridge to acknowledge GTP messages after some implementation dependent timeout. This allows the other side of the tunnel to release sent messages in a timely fashion, during a period when no messages are being sent in the opposite direction. If messages are being sent in the opposite direction there is no need to send this message as the synchronization occurs through the **gtp_header.last_seq_no_received** element of each sent message.

Special Notes

None

Forwardable

Yes - this GTP message can be encapsulated and sent in the **GTPForward** message. This will be used by either the Terminal Bridge or an old Access Bridge to acknowledge replies to forwarded GTP messages.

7.2.4 EstablishTunnelRequest Message

The **EstablishTunnelRequest** message has a message body containing the CDR encoded value of

```
union EstablishTunnelRequestBody switch (RequestType) {
    case INITIAL_REQUEST: InitialRequestBody initial_request_body;
    case RECOVERY_REQUEST: RecoveryRequestBody recovery_request_body;
    case NETWORK_REQUEST: NetworkRequestBody network_request_body;
    case TERMINAL_REQUEST: TerminalRequestBody terminal_request_body;
};
```

with the following definitions

```
typedef short RequestType;
const short INITIAL_REQUEST = 0;
const short RECOVERY_REQUEST = 1;
const short NETWORK_REQUEST = 2;
const short TERMINAL_REQUEST = 3;

struct InitialRequestBody {
    MobileTerminal::TerminalId terminal_id;
    MobileTerminal::HomeLocationAgent home_location_agent_reference;
    unsigned long time_to_live_request;
};

struct RecoveryRequestBody {
    MobileTerminal::TerminalId terminal_id;
    MobileTerminal::HomeLocationAgent home_location_agent_reference;
    struct LastAccessBridgeInfo {
        MobileTerminal::AccessBridge access_bridge_reference;
        unsigned long time_to_live_request;
        unsigned short last_seqno_received;
    } last_access_bridge_info;
    unsigned long time_to_live_request;
};

typedef RecoveryRequestBody NetworkRequestBody;
typedef RecoveryRequestBody TerminalRequestBody;
```

Source

Terminal Bridge

Description

This message is sent by the Terminal Bridge to establish or re-establish a tunnel with an Access Bridge. The **INITIAL_REQUEST** denotes that a new tunnel is requested. In tunnel re-establishment the new Access Bridge needs to know which re-establishment procedure to use:

- Access Recovery (see Section 8.4, “Access Recovery,” on page 8-11): **RECOVERY_REQUEST**
- Network Initiated Handoff (see Section 8.2, “Network Initiated Handoff,” on page 8-3): **NETWORK_REQUEST**
- Terminal Initiated Recovery (see Section 8.3, “Terminal Initiated Handoff,” on page 8-8): **TERMINAL_REQUEST**

The **terminal_id** and **home_location_agent_reference** will be used by the Access Bridge to accept or deny the request and to make the location update at the Home Location Agent of the terminal.

The **time_to_live_request** element is used to indicate the terminal’s desired life expectancy (in seconds) of this tunnel association upon should it be dropped.

Special Note

The **gtp_header**.**seq_no** and **gtp_header**.**last_seq_no_received** elements are always set to zero in this message.

Special Note

With regard to the various **time_to_live** parameters in all GTP messages, if the parameter is set to 0, then if sent by the terminal this indicates that the Access Bridge does not need to maintain any state or forward messages for a disconnected terminal. If sent by an Access Bridge, then the Access Bridge is indicating that it will not maintain any state and will not forward any messages for this terminal. In other words, the handoff will not be supported for this terminal.

Forwardable

No - this message cannot be encapsulated and sent via a **GTPForward** message.

7.2.5 EstablishTunnelReply Message

The **EstablishTunnelReply** message has a message body containing the CDR encoded value of

```
union EstablishTunnelReplyBody switch (ReplyType) {
    case INITIAL_REPLY: InitialReplyBody initial_reply_body;
    case RECOVERY_REPLY: RecoveryReplyBody recovery_reply_body;
    case NETWORK_REPLY: NetworkReplyBody network_reply_body;
    case TERMINAL_REPLY: TerminalReplyBody terminal_reply_body;
};
```

with the following definitions

```
typedef short ReplyType;
const short INITIAL_REPLY = 0;
const short RECOVERY_REPLY = 1;
const short NETWORK_REPLY = 2;
```

```
const short TERMINAL_REPLY = 3;

enum AccessStatus {
    ACCESS_ACCEPT,
    ACCESS_ACCEPT_RECOVERY,
    ACCESS_ACCEPT_HANDOFF,
    ACCESS_ACCEPT_LOCAL,
    ACCESS_REJECT_LOCATION_UPDATE_FAILURE,
    ACCESS_REJECT_ACCESS_DENIED,
    ACCESS_REJECT_RECOVERY_FAILURE
};

struct InitialReplyBody {
    AccessStatus status;
    MobileTerminal::AccessBridge access_bridge_reference;
    unsigned long time_to_live_reply;
};

struct RecoveryReplyBody {
    AccessStatus status;
    MobileTerminal::AccessBridge access_bridge_reference;
    struct OldAccessBridgeInfo {
        unsigned long time_to_live_reply;
        unsigned short last_seqno_received;
    } old_access_bridge_info;
    unsigned long time_to_live_reply;
};

typedef RecoveryReplyBody NetworkReplyBody;
typedef RecoveryReplyBody TerminalReplyBody;
```

Source

Access Bridge

Description

This message is sent by the Access Bridge in response to an **EstablishTunnelRequest** message. The status element has the following possible values:

- **ACCESS_ACCEPT**: in **InitialReplyBody**, indicates the successful establishment of a new tunnel; not used in **RecoveryReplyBody**.
- **ACCESS_ACCEPT_RECOVERY**: in **RecoveryReplyBody** it indicates the successful re-establishment of an old tunnel to the old Access Bridge; not used in **InitialReplyBody**.
- **ACCESS_ACCEPT_HANDOFF**: in **RecoveryReplyBody** it indicates the successful re-establishment of an old tunnel to a new Access Bridge; not used in **InitialReplyBody**.

- **ACCESS_ACCEPT_LOCAL**: in **InitialReplyBody**, indicates acceptance of access without location update at HLA (so called homeless terminal).
- **ACCESS_REJECT_LOCATION_UPDATE_FAILURE**: The location update at the Home Location Agent failed and the Access Bridge does not support homeless terminals.
- **ACCESS_REJECT_ACCESS_DENIED**: Access was denied by the Access Bridge. Generic reason. May be sent if a connection bridge is out of resources and cannot accept any more Tunnels.
- **ACCESS_REJECT_RECOVERY_FAILURE**: The Access Bridge did not get the information needed in the recovery from the old Access Bridge.

The **ACCESS_ACCEPT_RECOVERY** status indicates that the tunnel was established to the same Access Bridge as the last time a tunnel was established for this terminal. The Access Bridge will immediately set its next GTP header **gtp_header**.**seq_no** to the next to the value of the **last_access_bridge_info**.**last_seqno_received** element obtained in the **EstablishTunnelRequest** message, and will re-send any GTP messages lost when the tunnel was dropped. Likewise, the Terminal must immediately set its next GTP header **gtp_header**.**seq_no** to the next to the value of the **old_access_bridge_info**.**last_seqno_received** element of the **RecoveryReplyBody**, and will re-send any GTP messages lost when the tunnel was dropped.

If the tunnel was established to a new Access Bridge, then the Terminal Bridge should use the **old_access_bridge_info**.**last_seqno_received** element to indicate if any GTP messages sent by the terminal were lost by the old Access Bridge during a non-graceful handoff, and re-send them via **GTPForward** messages.

The **time_to_live_reply** element (not the **old_access_bridge_info**.**time_to_live_reply** element) is used to indicate the Access Bridge’s agreed to life expectancy of this tunnel association, and will be less than or equal to the terminal’s requested time to live.

Special Note

The **gtp_header**.**seq_no** and **gtp_header**.**last_seq_no_received** elements are always set to zero in this message.

Forwardable

No

7.2.6 ReleaseTunnelRequest Message

The **ReleaseTunnelRequest** message has a message body containing the CDR encoded value of

```
struct ReleaseTunnelRequestBody {
    unsigned long time_to_live;
};
```

Source

Terminal Bridge and Access Bridge

Description

This message may be sent by either the Terminal Bridge or the Access Bridge to gracefully tear down a tunnel. If sent by the Terminal Bridge, the **time_to_live** represents the time it desires the Access Bridge to maintain connections and forward outstanding GIOP messages for this terminal. If sent by the Access Bridge, then this **time_to_live** parameter represents the time it is willing to continue to forward GIOP messages for this terminal.

The sender of this message will send no more GTP messages directly on this tunnel. And will wait until it receives the reply before releasing the transport connectivity. The sender of this message will initiate the tear down of the transport connectivity after receipt of the reply.

Special Notes

None

Forwardable

No

7.2.7 ReleaseTunnelReply Message

The **ReleaseTunnelRequest** message has a message body containing the CDR encoded value of

```
struct ReleaseTunnelReplyBody {
    unsigned long time_to_live;
};
```

Source

Terminal Bridge and Access Bridge

Description

This message is sent by either the Terminal Bridge or the Access Bridge to acknowledge the graceful tear down of a tunnel. The **time_to_live** sent in this message must be less than or equal to the **time_to_live** sent in the **ReleaseTunnelRequest** message. If sent by the terminal, the **time_to_live** parameter represents the time it desires the Access Bridge to maintain connections and

forward outstanding GIOP messages for this terminal. If sent by the Access Bridge, then this **time_to_live** parameter represents the time its willing to continue to forward GIOP messages for this terminal.

The sender of this message will send no more GTP messages directly on this tunnel.

Upon sending or receiving this message, each end of the tunnel (Terminal and Access Bridge) may begin silently tearing down GIOP connections upon which there are no outstanding GIOP request messages.

The tunnel association for this terminal will be set to **inactive_forwarding** if the negotiated **time_to_live** is non-zero, and set to disconnected (and/or deleted) if **time_to_live** was negotiated to zero.

Special Notes

None

Forwardable

No

7.2.8 HandoffTunnelRequest Message

The **HandoffTunnelRequest** message has a message body containing the CDR encoded value of

```
struct HandoffTunnelRequestBody {
    MobileTerminal::AccessBridgeTransportAddressList
        new_access_bridge_transport_address_list;
};
```

Source

Access Bridge

Description

This message is sent by the Access Bridge to the Terminal Bridge in the network initiated handoff described in Section 8.2, "Network Initiated Handoff," on page 8-3.

The Terminal Bridge will use the **new_access_bridge_transport_address_list** to attempt to establish a tunnel to a new Access Bridge.

The sender of this message will send no more GTP messages directly on this tunnel until it received a **HandoffTunnelReply** message or times out after some implementation specific timeout waiting for the Terminal to establish a new Access Bridge. If it times out, then the Access Bridge may send a **ReleaseTunnelRequest** message to begin gracefully tearing down the tunnel. It will however continue to accept GTP messages sent by the Terminal Bridge and will hold them to either discard or process dependent upon the success or failure of the handoff.

The tunnel association for this terminal will be set to **handoff_in_progress** until receipt of a **HandoffTunnelReply** message.

Special Notes

None

Forwardable

No

7.2.9 HandoffTunnelReply Message

The **HandoffTunnelReply** message has a message body containing the CDR encoded value of

```
struct HandoffTunnelReplyBody {
    MobileTerminal::HandoffStatus status;
};
```

Source

Terminal Bridge

Description

This message is sent by the Terminal Bridge in response to **HandoffTunnelRequest** message.

If the Terminal Bridge successfully established a new **AccessBridge**, then status is set to **HANDOFF_SUCCESS**. The Terminal Bridge sends a **ReleaseTunnelRequest** message to the Access Bridge and waits for **ReleaseTunnelReply** message from the Access Bridge.

If the terminal does not support "make-before-break," then the Terminal Bridge should not try to establish connectivity to a new Access Bridge but to send a **HandoffTunnelReply** with status set to **NO_MAKE_BEFORE_BREAK**. The Terminal Bridge sends a **ReleaseTunnelRequest** message to the Access Bridge and waits for a **ReleaseTunnelReply** message from the Access Bridge. After that the Terminal Bridge establishes a tunnel to a new Access Bridge (see Section 8.2.5, "Alternative Handoff Procedure," on page 8-6).

If the terminal could not establish a tunnel to a new Access Bridge, then it will return a **HANDOFF_FAILURE** status in this message. The tunnel will then remain open and active until released by either endpoint via the **ReleaseTunnelRequest** / **ReleaseTunnelReply** sequence.

Special Notes

None

Forwardable

No

7.2.10 OpenConnectionRequest Message

The **OpenConnectionRequest** message has a message body containing the CDR encoded value of

```
struct OpenConnectionRequestBody {
    GIOP::TargetAddress target_object_reference;
    unsigned long open_connection_request_id;
    unsigned long timeout;
};
```

Source

Terminal Bridge and Access Bridge

Description

This message is sent by either the Terminal Bridge or the Access Bridge to allocate a connection on the remote end of the tunnel. To avoid allocation conflicts, Access Bridge uses even numbers and Terminal Bridge uses odd numbers (but not 0xFFFFFFFF, which is reserved as an error indicator; see Section 7.2.11, "OpenConnectionReply Message," on page 7-12). The **open_connection_request_id** will be returned in the **OpenConnectionReply** message. This handle is used so that the **target_object_reference** does not need to be returned in the **OpenConnectionReply** message.

The **target_object_reference** will be used by the receiver to connect to the target object.

The timeout is sent as an indication to the receiver of the sender's desired connection timeout. The receiver should return an error if this connection cannot be established within this period. Note that this timeout is by definition approximate because it does not take into account the transmission time of the request message.

Special Notes

None

Forwardable

No - new connections should be made through the current Access Bridge.

7.2.11 OpenConnectionReply Message

The **OpenConnectionReply** message has a message body containing the CDR encoded value of

```
struct OpenConnectionReplyBody {
    unsigned long open_connection_request_id;
    OpenConnectionStatus status;
    unsigned long connection_id; // 0xFFFFFFFF indicates failure
};

enum OpenConnectionStatus {
    OPEN_SUCCESS,
    OPEN_FAILED_UNREACHABLE_TARGET,
    OPEN_FAILED_OUT_OUT_RESOURCES,
    OPEN_FAILED_TIMEOUT,
    OPEN_FAILED_UNKNOWN_REASON
};
```

Source

Terminal Bridge and Access Bridge

Description

This message is sent by either the Terminal Bridge or the Access Bridge in response to an **OpenConnectionRequest** message. The **open_connection_request_id** element is the same as that passed in the **OpenConnectionRequest** message for which this is a reply. If a connection was established, the **connection_id** (allocated by the receiver of the **OpenConnectionRequest** message) is returned, and status is set to **OPEN_SUCCESS**. To avoid allocation conflicts, Access Bridges use even numbers and Terminal Bridges use odd numbers (but not 0xFFFFFFFF, which is reserved as an error indicator; see next paragraph).

If the connection could not be established within the requested time period, then the **connection_id** is set to **0xFFFFFFFF** and the status element is used to relay the failure reason.

Special Notes

None

Forwardable

Yes - this is due to the fact that outstanding **OpenConnectionRequests** may have been in progress during a transition to a new Access Bridge. However, if the new connection has no outstanding messages on it, then it should be closed and a **connection_id** = **0xFFFFFFFF** returned in this forwarded message with status = **OPEN_FAILED_TIMEOUT**.

7.2.12 CloseConnectionRequest Message

The **OpenConnectionRequest** message has a message body containing the CDR encoded value of

```
struct CloseConnectionRequestBody {
    unsigned long connection_id; // 0xFFFFFFFF denotes all connections
for sender
};
```

Source

Terminal Bridge and Access Bridge

Description

This message is sent by either the Terminal Bridge or the Access Bridge to close a currently open connection. If the **connection_id** is set to **0xFFFFFFFF**, then all connections associated with this Tunnel should be closed.

Special Notes

None

Forwardable

Yes - this will be used by either the Terminal Bridge or an old Access Bridge to gracefully shut down open GIOP connections after a terminal has moved to a new Access Bridge.

7.2.13 CloseConnectionReply Message

The **CloseConnectionReply** message has a message body containing the CDR encoded value of

```
struct CloseConnectionReplyBody {
    unsigned long connection_id; // same as in request
    CloseConnectionStatus status;
};

enum CloseConnectionStatus {
    CLOSE_SUCCESS,
    CLOSE_FAILED_INVALID_CONNECTION_ID,
    CLOSE_FAILED_UNKNOWN_REASON
};
```

Source

Terminal Bridge and Access Bridge

Description

This message is sent by either the Terminal Bridge or the Access Bridge in response to a **CloseConnectionRequest** message. The **connection_id** element is the same as is sent in the **CloseConnectionRequest** message for which this is a reply.

Special Notes

None

Forwardable

Yes - this will be used by either the Terminal or an old Access Bridge, to gracefully shut down open connections after a terminal has moved to a new Access Bridge.

7.2.14 ConnectionCloseIndication Message

The **ConnectionCloseIndication** message has a message body containing the CDR encoded value of

```
struct ConnectionCloseIndicationBody {
    unsigned long connection_id; // 0xFFFFFFFF means all connection for recipient
    ConnectionCloseReason reason;
};

enum ConnectionCloseReason {
    CLOSE_REASON_REMOTE_END_CLOSE,
    CLOSE_REASON_RESOURCE_CONSTRAINT,
    CLOSE_REASON_IDLE_CLOSED,
    CLOSE_REASON_TIME_TO_LIVE_EXPIRED,
    CLOSE_REASON_UNKNOWN_REASON
};
```

Source

Terminal Bridge and Access Bridge

Description

This message is sent by either the Terminal Bridge or the Access Bridge to alert the other end of the tunnel that a connection was asynchronously closed, (not in response to a **CloseConnectionRequest** message).

If all open connections for this tunnel association were closed, then the **connection_id** element will be set to **0xFFFFFFFF**.

The reason element is used to indicate the reason for the connection closure. The element field has the following meanings:

- **CLOSE_REASON_REMOTE_END_CLOSE**: The remote end of the GIOP connection closed the connection.
- **CLOSE_REASON_RESOURCE_CONSTRAINT**: The sender closed this connection because of a resource constraint.
- **CLOSE_REASON_IDLE_CLOSED**: The sender closed the connection after an implementation dependent timeout and after all outstanding GIOP requests had been completed and the connection could be safely closed.

- **CLOSE_REASON_TIME_TO_LIVE_EXPIRED**: The **time_to_live** for this terminal who had moved expired.

The receiver of this message should mark the indicated connections as deleted in its local data structures. If a **ConnectionCloseIndication** message is received for a **connection_id** not valid on the receiver (probably because the receiver had already deleted it locally), then the message will be silently discarded.

Special Notes

None

Forwardable

Yes - this will be used by either the Terminal Bridge or an old Access Bridge to indicate asynchronous connection closures after a terminal has moved to a new Access Bridge. This is used to indicate that the **time_to_live** has expired with the reason set to **CLOSE_REASON_TIME_TO_LIVE_EXPIRED**. It is also sent with the reason set to **CLOSE_REASON_IDLE_CLOSED** if all outstanding GIOP requests have been completed and the connection was safely closable.

7.2.15 GIOPData Message

The **GIOPData** message has a message body containing the CDR encoded value of

```
struct GIOPDataBody {
    unsigned long connection_id;
    unsigned long giop_message_id;
    MobileTerminal::GIOPEncapsulation giop_message;
};
```

Source

Terminal Bridge and Access Bridge

Description

This message is sent by either the Terminal Bridge or the Access Bridge and contains an encapsulated GIOP message. The **giop_message_id** element is assigned by the sending bridge. It is used by the receiving bridge in **GIOPDataError** message to indicate unsuccessful delivery of a GIOP message. The **connection_id** is the receiver's connection on which this message is to be sent.

Special Notes

If the delivery of the encapsulated GIOP message is successful, this success is not indicated explicitly to the sender of the **GIOPData** message. Instead, successful delivery is implicitly indicated by normal acknowledgment of the GTP sequence number of the **GIOPData** message.

Forwardable

Yes - this will be used by either the Terminal Bridge or an old Access Bridge to forward GIOP messages.

7.2.16 GIOPDataError Message

The **GIOPDataError** message has a message body containing the CDR encoded value of

```
struct GIOPDataErrorBody {
    unsigned long giop_message_id;
    DeliveryStatus status;
};

enum DeliveryStatus {
    DELIVERY_FAILED_INVALID_CONNECTION_ID,
    DELIVERY_FAILED_UNKNOWN_REASON
};
```

Source

Terminal Bridge and Access Bridge

Description

This message is sent by either the Terminal Bridge or the Access Bridge to indicate unsuccessful delivery of a GIOP message. The status element is set to the appropriate failure code.

Special Notes

None

Forwardable

Yes - this will be used by either the Terminal Bridge or an old Access Bridge to forward indications of unsuccessful delivery of a GIOP message.

7.2.17 GTPForward Message

The **GTPForward** message has a message body containing the CDR encoded value of

```
struct GTPForwardBody {
    MobileTerminal::AccessBridge access_bridge_reference;
        // source if sent by Access Bridge, destination if sent by Terminal Bridge
    unsigned long gtp_message_id;
    MobileTerminal::GTPEncapsulation gtp_message;
        // including GTP header
};
```

Source

Terminal Bridge and Access Bridge

Description

This message is sent by either the Terminal Bridge or the Access Bridge to forward messages to/from an old Access Bridge. The **gtp_message_id** is allocated by the receiver so that it can identify the GTP message in the **GTPForwardReply** message. This handle is used so that the **access_bridge_reference** does not need to be returned in the **GTPForwardReply** message.

If the message is sent by a Terminal, then the **gtp_from_terminal** operation will be invoked on the **access_bridge_reference** to forward the message to the "old" Access Bridge; see Section 8.5, "GTP Message Forwarding," on page 8-14.

If the message is sent by an Access Bridge, the **access_bridge_reference** will be the source of the forwarded GTP message.

Special Notes

None

Forwardable

No - a **GTPForward** message cannot be encapsulated in another **GTPForward** message. However, Access Bridges can forward forwarded messages given to them by invoking the **gtp_from_terminal** and **gtp_to_terminal** operations.

7.2.18 GTPForwardReply Message

The **GTPForwardReply** message has a message body containing the CDR encoded value of

```
struct GTPForwardReplyBody {
    unsigned long gtp_message_id;
    ForwardStatus status;
};

enum ForwardStatus {
    FORWARD_SUCCESS,
    FORWARD_ERROR_ACCESS_BRIDGE_UNREACHABLE,
    FORWARD_ERROR_UNKNOWN_SENDER,
    FORWARD_UNKNOWN_FORWARD_ERROR
};
```

Source

Terminal Bridge and Access Bridge

Description

This message is sent by either the Terminal Bridge or the Access Bridge in response to a **GTPForward** message. The **gtp_message_id** element is the same as passed in the **GTPForward** message for which this is a reply.

If this reply message is sent by an Access Bridge, the **FORWARD_SUCCESS** status indicates that the encapsulated GTP message was delivered to the old Access Bridge. Any needed GTP replies or GTP error messages will be returned in separate **GTPForward** messages from that Access Bridge. However, if the status is either **FORWARD_ERROR_ACCESS_BRIDGE_UNREACHABLE** or **FORWARD_ERROR_UNKNOWN_SENDER**, then the terminal should consider the tunnel on that access bridge to be lost.

If this reply message is sent by a Terminal Bridge, upon receipt of this message the Access Bridge will call back to the originating Access Bridge (by mapping **gtp_message_id** back to the **access_bridge_reference** and the **gtp_message_id** given through the **gtp_to_terminal** operation) by invoking its **gtp_acknowledge** operation to deliver the status field. The **FORWARD_SUCCESS** status indicates that the encapsulated GTP message was accepted by the Terminal GTP engine. If the Terminal has already forgotten about or given up on the Access Bridge who sent the forwarded GTP message, then the status will be set to **FORWARD_ERROR_UNKNOWN_SENDER**. The Access Bridge will then consider that terminal lost, and begin tearing down its tunnel end as if the **time_to_live** had expired.

Special Notes

None

Forwardable

Yes - this is due to the fact that outstanding **GTPForwardRequests** may have been in progress during a transition to a new Access Bridge.

7.2.19 Error Message

The Error message has a message body containing the CDR encoded value of

```
struct ErrorBody {
    unsigned short gtp_seq_no; // seq_no element in GTP header
    ErrorCode error_code;
};

enum ErrorCode {
    ERROR_UNKNOWN_SENDER,
    ERROR_PROTOCOL_ERROR,
    ERROR_UNKNOWN_FATAL_ERROR
};
```

Source

Terminal Bridge and Access Bridge

Description

This message is sent by either the Terminal Bridge or the Access Bridge to handle GTP protocol errors and to initiate a shutdown. The **gtp_header.seq_no** of the GTP message is provided for debugging purposes since this tunnel will be immediately destroyed.

Special Notes

None

Forwardable

Yes - this will be used by either the Terminal Bridge or an old Access Bridge to cause a disorderly shutdown since the Terminal Bridge and the old Access Bridge are obviously out of sync.

7.3 TCP Tunneling

In TCP Tunneling the GTP messages are transmitted in a byte stream without any padding or message boundary marker.

The transport end-point is given as a string: <ip_address>:port_number, where <ip_address> is either a DNS name of a host or an IP address in dotted decimal notation.

7.4 UDP Tunneling

In UDP Tunneling the GTP messages are transmitted using the framing protocol, called UDP Tunneling Protocol, described below, in the payload of UDP datagrams.

The transport end-point is given as a string: <ip_address>:<port_number>, where <ip_address> is an IP address in dotted decimal notation (123.45.67.89, for example) so that the terminal does not need to do a DNS lookup.

7.4.1 UDP Tunneling Protocol

The UDP Tunneling Protocol (UTP) provides the reliability and ordered delivery of messages assumed by the GIOP Tunneling Protocol. UTP assumes that it does not get corrupted data.

UTP defines encapsulation of GTP messages. It also supports segmentation and re-assembly of GTP messages and selective acknowledgments.

UTP is chunk-based in the sense that several GTP messages can be concatenated in one UTP message. A UTP message is the payload of a UDP datagram. A UTP message contains a UTP header and one or more UTP chunks.

The UTP header is four bytes: UTP Sequence Number (unsigned short) and Number of UTP chunks (unsigned short) in the UTP message. The network byte order (that is Big-Endian) is always used to express numeric values. In UTP, strings are always in 8-bit ANSI ASCII format.

The basic structure of a UTP chunk is TFLV: type-flags-length-value. However, some chunks do not have Flags, Length, and/or Value field.

- The Type field is one octet.
- If present, the Flags field is one octet. It is used to denote fragmentation.
- The Length field is 0-2 octets telling the length of the Value field in the network byte order if the Value field can be of variable length.
- The Value field if present contains the payload of a UTP chunk.

The UTP chunks are:

1. ***InitialAccessRequest***: sent by the Terminal Bridge. The Flags (one octet) and Length (unsigned short) fields are present. The Value (variable length) field contains a cookie (sequence of octets) and the transport address end-point of the Terminal Bridge (string).
2. ***InitialAccessReply***: sent by the Access Bridge. The Flags (one octet) and Length (unsigned short) fields are present. The Value (variable length) field contains a cookie (sequence of octets) and the transport address end-point of the Access Bridge (string).
3. ***Pause***: sent by the Terminal or Access Bridge. No Flags, Length, and Value field. The receiving bridge should interpret this chunk so that the sending bridge will silently discard all UTP messages until it receives the Resume chunk.
4. ***Resume***: sent by the Terminal or Access Bridge. No Flags, Length, and Value field. The receiving bridge should interpret this chunk so that the sending bridge will start to accept the UTP chunks again.
5. ***Acknowledge***: sent by the Terminal or Access Bridge. No Flag Field. The Length (one octet) tells the number of entries in the Value field. The actual length of the Value field in octets is the content of the Length field multiplied by two. The first unsigned short tells the highest Sequence Number of UTP messages received in order. The rest unsigned shorts tell which other UTP messages have been received.
6. ***GTPData***: sent by the Terminal or Access Bridge. Flags (one octet) indicate fragmentation. The Length field (unsigned short) tells the length of the Value field.
7. ***CloseRequest***: sent by the Terminal or Access Bridge. No Flags, Lenght, and Value field.
8. ***CloseReply***: sent by the Terminal or Access Bridge. No Flags, Lenght, and Value field.
9. ***CloseIndication***: sent by the Terminal or Access Bridge. No Flags, Lenght, and Value field.

7.4.2 Sequence Numbering

Each communication session must start sequence numbering from 0x0001. In other words the UTP message containing the InitialAccessRequest chunk (or its first fragment) and the associated reply containing the InitialAccessReply chunk (or its first fragment) MUST have UTP Sequence Number 0x0001.

The sequence number counting follows the usual modulo arithmetic with the exception that the sequence number 0x0001 follows the sequence number 0xFFFF.

The UTP Sequence Number 0x0000 is reserved for UTP messages that only contain the Acknowledge chunk and/or the Pause chunk. These messages MUST NOT be acknowledged.

7.4.3 Retransmission Policy

Retransmissions in UDP Tunneling Protocol are controlled by selective acknowledgements by retransmission timers as in TCP with SACK option enabled.

When the sender detects from the selective acknowledgements that the acknowledged sequence has holes, the missing messages are immediately retransmitted.

A message is also retransmitted when its retransmission timer expires. The retransmission timer MUST be computed as specified for TCP in RFC 2988 [RFC 2988].

7.4.4 Fragmentation

The two rightmost bits of the Flags field are used to denote fragmentation of the Value field:

- 0x00: middle segment

- 0x01: first segment

- 0x02: last segment

- 0x03: unfragmented chunk

7.4.5 InitialAccessRequest

The chunk Type is 0x01. The Flags field (one octet) indicate fragmentation. The Length field is two octets indicating the length of the Value field as an unsigned short.

The Value field contains CDR encoded value of

```
struct InitialAccessRequestChunk {
    sequence<octet> cookie;
    string terminal_bridge_udp_address;
};
```

where **cookie** is some bit-pattern selected by the Terminal Bridge and **terminal_bridge_udp_address** is a string containing the IP address (in dotted decimal notation) of the terminal and the UDP port number to which the Access Bridge shall send the UTP messages ("123.45.67.89:9876", for example).

The **InitialAccessRequest** chunk can only be sent by the Terminal Bridge.

Note: UTP message containing this chunk (or its first fragment) MUST have sequence number 0x0001.

7.4.6 InitialAccessReply

The chunk Type is 0x02. The Flags field (one octet) indicate fragmentation. The Length field is two octets indicating the length of the Value field as an unsigned short.

The Value field contains CDR encoded value of

```
struct InitialAccessReplyChunk {
    sequence<octet> cookie;
    string access_bridge_udp_address;
}
```

where **cookie** is the bit-pattern received in the **InitialAccessRequest** from the Terminal Bridge and **access_bridge_udp_address** is a string containing the IP address (in dotted decimal notation) of the Access Bridge and the UDP port number to which the Terminal Bridge shall send the UTP messages.

The **InitialAccessReply** chunk can only be sent by the Access Bridge.

Note: UTP message containing this chunk (or its first fragment) MUST have sequence number 0x0001.

7.4.7 Pause

The chunk Type is 0x03. The chunk does not have other field.

The receiving bridge should interpret this chunk so that the sending bridge will silently discard all UTP messages until it sends the Resume chunk.

Both Access and Terminal Bridge can use this chunk.

Note 1: If the UTP message contains only the Pause chunk (with or without the Acknowledge chunk), then the UTP Sequence Number MUST be 0x0000. The receiver SHOULD NOT acknowledge such a message.

Note 2: After sending the Pause chunk, the sender SHOULD reply to each arriving message by a UTP message containing the Pause chunk. The UTP message MAY also contain Acknowledge and/or GTPData chunks.

7.4.8 Resume

The chunk Type is 0x04. The chunk does not have other fields.

The receiving bridge should interpret this chunk so that the sending bridge will accept UTP messages again.

Both Access and Terminal Bridge can use this chunk.

Note: The Resume chunk SHOULD be included in each UTP message until the sender has received an acknowledgement for a message containing a Resume chunk.

7.4.9 Acknowledge

The chunk Type is 0x05. The chunk does not have the Flags field. The Length (one octet) tells the number of entries in the Value field. The actual length of the Value field in octets is the content of the Length field multiplied by two.

The first unsigned short in the Value field tells the highest Sequence Number of UTP messages received in order. The rest unsigned shorts tell which other UTP messages have been received.

Both Access and Terminal Bridge can use this chunk.

Note: If the UTP message contains only the Acknowledge chunk (with or without the Pause chunk), then the UTP Sequence Number MUST be 0x0000. The receiver SHOULD NOT acknowledge such a message.

7.4.10 GTPData

The chunk Type is 0x06. The Flags field (one octet) indicates fragmentation. The Length field (unsigned short) tells the length of the Value field.

The Value field contains a GTP message or a part of it.

7.4.11 Close Request

The chunk Type is 0x07. The chunk does not have any other fields.

The chunk can be sent either by the Terminal Bridge or by the Access Bridge.

The bridge sends this chunk when it wants to close the communication session. After sending this chunk the bridge mwaits for the UTP message cointaining the CloseReply chunk. The bridge sending the CloseRequest chunk MUST process and acknowledge all UTP messages until receiving the CloseReply chunk.

When a bridge receives the CloseRequest chunk, it SHOULD immediately retransmit all unacknowledged UTP messages as well as all remaining GTP messages before sending the CloseReply chunk.

7.4.12 CloseReply

The chunk Type is 0x08. The chunk does not have any other fields.

The chunk can be sent either by the Terminal Bridge or by the Access Bridge in response to the CloseRequest chunk.

The bridge receiving the CloseRequest chunk SHOULD NOT send the CloseReply chunk before all other UTP messages have been acknowledgeged. After sending the CloseReply chunk, the bridge SHOULD wait for the acknowledgement before releasing all data structures associated with the UTP communication session.

The bridge receiving the CloseReply chunk SHOULD acknowledge the UTP message containing the CloseReply chunk by a UTP message having sequence number 0x0000. After sending this reply the bridge can release all data structures associated to the UTP communication session.

If a bridge receives CloseReply chunk, but it has not send the CloseRequest chunk, the bridge SHOULD send the CloseIndication chunk. See section 7.4.n 'CloseIndication'.

7.4.13 CloseIndication

The chunk Type is 0x09. The chunk does not have any other fields.

The chunk can be sent either by the Terminal Bridge or by the Access Bridge.

The bridge send this chunk to inform the peer bridge that it will close the communication session. The bridge MAY wait for the acknowledgement of the UTP message containing the CloseIndication chunk. In this case the bridge SHOULD discard all arriving UTP messages and response by retransmitting the UTP message containing the CloseIndication chunk. However, the bridge does not need to wait for the acknowledgement of the UTP message containing the CloseIndication chunk. It can immediately, after sending the CloseIndication chunk, release all data structures related to this communication session and stop accepting messages to the UDP port.

When the peer bridge receives the CloseIndication chunk, it SHOULD immediately acknowledge that UTP message and stop using the UDP end-point of the peer bridge. After sending the acknowledgement, the bridge SHOULD release all data structures related to the UTP communication session.

7.5 WAP Tunneling

The WAP Tunneling Protocol (WAPTP) uses the Wireless Application Protocol (WAP) to transmit GTP messages between Terminal and Access Bridge.

The main design principle in WAPTP has been simplicity of the implementation. It is assumed that WAPTP will be used in small embedded devices with limited capabilities.

WAPTP ensures that the assumptions stated by GTP are not violated, specifically that no corrupted data is delivered and that the order of GTP messages is preserved.

7.5.1 Wireless Datagram Protocol

WAPTP uses the Wireless Datagram Protocol (WDP) [WDP] of the WAP specification. It operates above the data capable bearer services supported by multiple network types. WDP specification describes reference models for a wide variety of networks.

WDP provides a service similar to UDP, such as unreliable transmission of datagrams and use of port numbers to identify multiple applications in one transport address.

> *"WDP supports several simultaneous communication instances from a higher layer over a single underlying WDP bearer service. The port number identifies the higher layer entity above WDP."* [WDP, 5.2]
>
> *"The services offered by WDP include application addressing by port numbers, optional segmentation and reassembly and optional error detection. The services allow for applications to operate transparently over different available bearer services."* [WDP, 5.1]

If the used bearer does not provide segmentation and reassembly (SAR), then it is the responsibility of the WDP implementation to do it.

> *"If the underlying bearer does not provide Segmentation and Reassembly the feature is implemented by the WDP provider in a bearer dependent way."* [WDP, 7.1]

The maximum size of datagram is bearer dependent. It is assumed that the GTP implementation does not attempt to send GTP messages that are larger than the maximum datagram size for given bearer (this implies that the ORB also knows this limitation and fragments GIOP messages accordingly).

WDP ensures the correct order of datagram segments, but not the order of datagrams themselves.

7.5.2 WAP Tunneling Protocol

In WAPTP, GTP messages are transmitted in Invoke PDUs of WAP WDP, one GTP message in one WDP datagram.

WDP datagrams are not guaranteed to preserve order, so WAPTP MUST delay the delivery of GTP messages that have higher sequence numbers than expected.

7.5.3 WAPTP address types

The WDP supports several address types including IP addresses (both IPv4 and IPv6), MSISD (a telephone number) in various flavors (IS_637, ANSI_136, GSM, CDMA, iDEN, FLEX, TETRA), GSM_Service_Code, TETRA_ISI, and Mobitex MAN. The WDP transport address end-points are given as CDR encapsulation of

```
struct WDPAddressFormat {
    octet wdp_version;// mostly 0x00, depends on bearer; see [WDP]
    octet wap_assigned_number;// identifies network, bearer, address
                                    // type combination; see [WDP, Appendix C]
    unsigned short wap_port;// Port number
```

```
    string address;
};
```

The most usual address types are IP address and telephone number (MSISDN). An IP address must be in the decimal dotted notation (e.g., 123.1.2.23) so that the terminal does not need to make a DNS lookup. All possible stringified formats of telephone numbers are specified in [GFD].

7.6 Bluetooth Tunneling

Because the purpose of tunneling is just to deliver GIOP messages over wireless links, tunneling should be done as low as possible in the Bluetooth stack (Figure 7-2) to have minimum overhead.

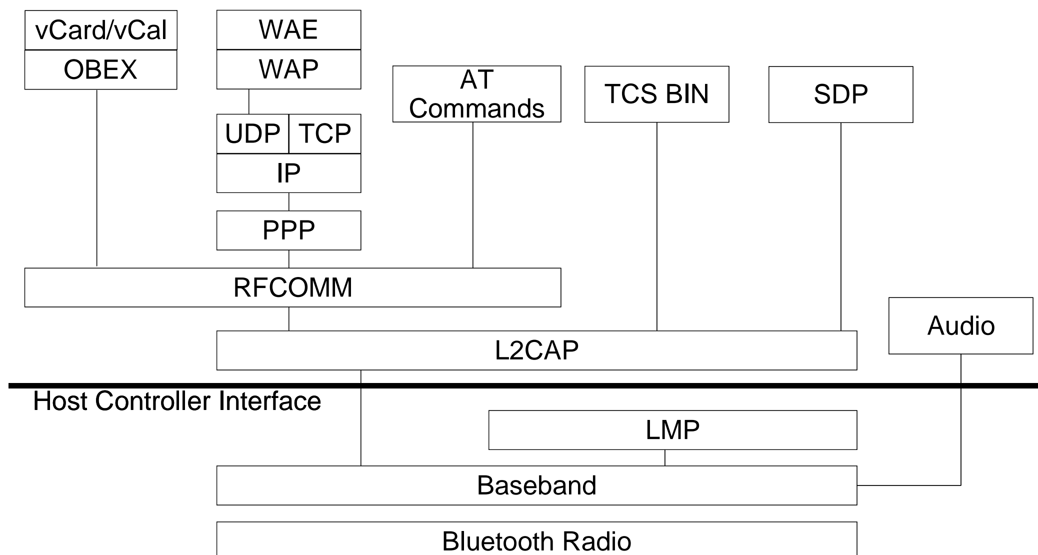

Figure 7-2 Bluetooth Protocol Stack [BT-SIG]

Any Bluetooth profile, or even the RFCOMM protocol, is not appropriate because they have many additional features that are not needed for tunneling. The Baseband protocol through the Host Controller Interface (HCI) is not sufficient because it does not support protocol multiplexing and de-multiplexing for upper layers. Therefore, L2CAP is the most suitable protocol in the Bluetooth stack to be used in GIOP Tunneling.

Since L2CAP is right above HCI, it has a low overhead but still provides protocol multiplexing and de-multiplexing for upper layers. L2CAP provides connection-oriented data services, a reliable channel and ordered delivery of messages using the mechanisms available at the Baseband layer. It also provides notification of disorderly connection lost. However, L2CAP have limits for packet size, so GTP message segmentation and reassembly MUST be implemented in order to provide a possibility to send messages of any size.

In Bluetooth Tunneling the GTP messages are transmitted using L2CAP Tunneling Protocol (LTP) in the payload of L2CAP packets.

The transport end-point is given as a string: <BD_ADDR>#<PSM>, where <BD_ADDR> is a unique 48-bit Bluetooth device address given in coloned hexadecimal notation (e.g., 7F:00:00:01:05:B3) and <PSM> is protocol/service multiplexer given as an unsigned integer in range 0...65535 (two octets).

7.6.1 LTP Tunneling Protocol

The L2CAP Tunneling Protocol (LTP) provides the reliability and ordered delivery of messages assumed by the GIOP Tunneling Protocol. LTP assumes that is does not get corrupted data.

LTP defines encapsulation of GTP messages. It also supports segmentation and reassembly of GTP messages.

A LTP message is the payload of a L2CAP packet. One LTP message contains either one GTP message or a fragment of one GTP message. The structure of a LTP message is FLV: flags-length-value. The network byte order (that is Big-Endian) is always used to express numeric values.

- The Flags field is one octet. It is used to denote fragmentation (segmentation).
- The Length field is 2 octets telling the length of the Value field in the network byte order (that is Big-Endian).
- The Value field contains the LTP payload, that is one GTP message or a fragment of a GTP message.

7.6.2 Fragmentation

The two rightmost bits of the Flags field is used to denote fragmentation of the Value field:

- 0x00: middle segment

- 0x01: first segment

- 0x02: last segment

- 0x03: unfragmented message