Wireless Access and Terminal Mobility in CORBA Specification

Finalized Specification
September 2002

PATENT

NOTICE

ISSUE REPORTING

1. **Overview 1-1**
   1.1 Design Rationale 1-1
   1.2 Proof of Concept 1-2
   1.3 References 1-2
2. **Architectural Framework 2-1**
   2.1 Key Concepts 2-1
   2.2 Overall Architecture 2-2
3. **Mobile IOR 3-1**
   3.1 IIOP Profiles in Mobile IOR 3-1
      3.1.1 Address information in IIOP Profiles in Mobile IORs 3-2
      3.1.2 Mobile Object Key Format 3-2
   3.2 The Mobile Terminal Profile 3-2
      3.2.1 Mobile Terminal Profile Structure 3-3
      3.2.2 TAG_HOME_LOCATION_INFO Component 3-4
   3.3 Translation to Mobile Target Object 3-4
   3.4 Interoperability with GIOP 1.0 and 1.1 3-5
   3.5 Additional Type Definitions 3-5
4. **Home Location Agent 4-1**
   4.1 Location Update 4-1
   4.2 Discovery 4-4
   4.3 Message Processing 4-4
   4.4 Terminal Ids 4-5
5. **Access Bridge 5-1**
   5.1 Discovery 5-1
   5.2 Query 5-2
   5.3 Message Processing 5-2
   5.4 Mobility Event Notifications 5-3
6. **Terminal Bridge 6-1**
   6.1 Mobility Event Notifications 6-1
7. **GIOP Tunneling 7-1**
   7.1 Tunnel Establishment 7-2
      7.1.1 Transport End-Point Detection 7-2
   7.2 GIOP Tunneling Protocol 7-2

7.2.1 GTP Message Structure 7-3
7.2.2 GTP Messages 7-3
7.2.3 IdleSync Message 7-4
7.2.4 EstablishTunnelRequest Message 7-5
7.2.5 EstablishTunnelReply Message 7-6
7.2.6 ReleaseTunnelRequest Message 7-8
7.2.7 ReleaseTunnelReply Message 7-9
7.2.8 HandoffTunnelRequest Message 7-10
7.2.9 HandoffTunnelReply Message 7-11
7.2.10 OpenConnectionRequest Message 7-11
7.2.11 OpenConnectionReply Message 7-12
7.2.12 CloseConnectionRequest Message 7-13
7.2.13 CloseConnectionReply Message 7-14
7.2.14 ConnectionCloseIndication Message 7-14
7.2.15 GIOPData Message 7-16
7.2.16 GIOPDataErroe Message 7-16
7.2.17 GTPForward Message 7-17
7.2.18 GTPForwardReply Message 7-18
7.2.19 Error Message 7-19
7.3 TCP Tunneling 7-20
7.4 UDP Tunneling 7-20
7.4.1 UDP Tunneling Protocol 7-20
7.4.2 Fragmentation 7-21
7.4.3 InitialAccessRequest 7-21
7.4.4 InitialAccessReply 7-22
7.4.5 Pause 7-22
7.4.6 Resume 7-22
7.4.7 Acknowledge 7-22
7.4.8 GTPData 7-23
7.5 WAP Tunneling 7-23
7.5.1 Wireless Datagram Protocol 7-23
7.5.2 WAP Tunneling Protocol 7-24
7.5.3 WAPTP address types 7-24

8. Handoff and Access Recovery 8-1

8.1 Initiation 8-2
8.2 Network Initiated Handoff 8-3
8.2.1 Old Access Bridge 8-3
8.2.2 New Access Bridge 8-4
8.2.3 Terminal Bridge 8-5

8.2.4 Message Sequence Chart 8-6
8.2.5 Alternative Handoff Procedure 8-6
8.2.6 IDL 8-7

8.3 Terminal Initiated Handoff 8-8
8.3.1 Terminal Bridge 8-8
8.3.2 New Access Bridge 8-9
8.3.3 Old Access Bridge 8-9
8.3.4 Message Sequence Chart 8-10
8.3.5 IDL 8-10

8.4 Access Recovery 8-11
8.4.1 Recovery to the Old Access Bridge 8-11
8.4.2 Recovery to New Access Bridge 8-12

8.5 GTP Message Forwarding 8-14

8.6 Terminal Tracking 8-17

9. OMG IDL A-1

10. Mandatory /Optional Requirements B-1
10.0.1 Home Location Agent B-1
10.0.2 Access Bridge B-1
10.0.3 Terminal Bridge B-2

10.1 Optional Requirements B-2
10.1.1 Home Location Agent B-2
10.1.2 Access Bridge B-2
10.1.3 Terminal Bridge B-2

Preface

About the Object Management Group

The Object Management Group, Inc. (OMG) is an international organization supported by over 800 members, including information system vendors, software developers and users. Founded in 1989, the OMG promotes the theory and practice of object-oriented technology in software development. The organization's charter includes the establishment of industry guidelines and object management specifications to provide a common framework for application development. Primary goals are the reusability, portability, and interoperability of object-based software in distributed, heterogeneous environments. Conformance to these specifications will make it possible to develop a heterogeneous applications environment across all major hardware platforms and operating systems.

OMG's objectives are to foster the growth of object technology and influence its direction by establishing the Object Management Architecture (OMA). The OMA provides the conceptual infrastructure upon which all OMG specifications are based.

What is CORBA?

The Common Object Request Broker Architecture (CORBA), is the Object Management Group's answer to the need for interoperability among the rapidly proliferating number of hardware and software products available today. Simply stated, CORBA allows applications to communicate with one another no matter where they are located or who has designed them. CORBA 1.1 was introduced in 1991 by Object Management Group (OMG) and defined the Interface Definition Language (IDL) and the Application Programming Interfaces (API) that enable client/server object interaction within a specific implementation of an Object Request Broker (ORB). CORBA 2.0, adopted in December of 1994, defines true interoperability by specifying how ORBs from different vendors can interoperate.

OMG Documents

The OMG documentation is organized as follows:

OMG Modeling

- ***Unified Modeling Language (UML) Specification*** defines a graphical language for visualizing, specifying, constructing, and documenting the artifacts of distributed object systems.
- ***Meta-Object Facility (MOF) Specification*** defines a set of CORBA IDL interfaces that can be used to define and manipulate a set of interoperable metamodels and their corresponding models.
- ***OMG XML Metadata Interchange (XMI) Specification*** supports the interchange of any kind of metadata that can be expressed using the MOF specification, including both model and metamodel information.

Object Management Architecture Guide

This document defines the OMG's technical objectives and terminology and describes the conceptual models upon which OMG standards are based. It defines the umbrella architecture for the OMG standards. It also provides information about the policies and procedures of OMG, such as how standards are proposed, evaluated, and accepted.

CORBA: Common Object Request Broker Architecture and Specification

Contains the architecture and specifications for the Object Request Broker.

OMG Interface Definition Language (IDL) Mapping Specifications

These documents provide a standardized way to define the interfaces to CORBA objects. The IDL definition is the contract between the implementor of an object and the client. IDL is a strongly typed declarative language that is programming language-independent. Language mappings enable objects to be implemented and sent requests in the developer's programming language of choice in a style that is natural to that language. The OMG has an expanding set of language mappings, including Ada, C, C++, COBOL, IDL to Java, Java to IDL, Lisp, and Smalltalk.

CORBAservices

Object Services are general purpose services that are either fundamental for developing useful CORBA-based applications composed of distributed objects, or that provide a universal-application domain-independent basis for application interoperability.

These services are the basic building blocks for distributed object applications. Compliant objects can be combined in many different ways and put to many different uses in applications. They can be used to construct higher level facilities and object frameworks that can interoperate across multiple platform environments.

Adopted OMG Object Services are collectively called CORBAservices and include specifications such as *Collection*, *Concurrency*, *Event*, *Externalization*, *Naming*, *Licensing*, *Life Cycle*, *Notification*, *Persistent Object*, *Property*, *Query*, *Relationship*, *Security*, *Time*, *Trader*, and *Transaction*.

CORBAfacilities

Common Facilities are interfaces for horizontal end-user-oriented facilities applicable to most domains. Adopted OMG Common Facilities are collectively called CORBAfacilities and include specifications such as *Internationalization and Time*, and *Mobile Agent Facility*.

Object Frameworks and Domain Interfaces

Unlike the interfaces to individual parts of the OMA "plumbing" infrastructure, Object Frameworks are complete higher level components that provide functionality of direct interest to end-users in particular application or technology domains.

Domain Task Forces concentrate on Object Framework specifications that include Domain Interfaces for application domains such as Finance, Healthcare, Manufacturing, Telecoms, E-Commerce, and Transportation.

Currently, specifications are available in the following domains:

- *CORBA Business*: Comprised of specifications that relate to the OMG-compliant interfaces for business systems.
- *CORBA Finance*: Targets a vitally important vertical market: financial services and accounting. These important application areas are present in virtually all organizations: including all forms of monetary transactions, payroll, billing, and so forth.
- *CORBA Healthcare*: Comprised of specifications that relate to the healthcare industry and represents vendors, healthcare providers, payers, and end users.
- *CORBA Manufacturing*: Contains specifications that relate to the manufacturing industry. This group of specifications defines standardized object-oriented interfaces between related services and functions.
- *CORBA Telecoms*: Comprised of specifications that relate to the OMG-compliant interfaces for telecommunication systems.
- *CORBA Transportation*: Comprised of specifications that relate to the OMG-compliant interfaces for transportation systems.

Obtaining OMG Documents

The OMG collects information for each book in the documentation set by issuing Requests for Information, Requests for Proposals, and Requests for Comment and, with its membership, evaluating the responses. Specifications are adopted as standards only when representatives of the OMG membership accept them as such by vote. (The policies and procedures of the OMG are described in detail in the *Object Management Architecture Guide*.)

OMG formal documents are available from our web site in PostScript and PDF format. To obtain print-on-demand books in the documentation set or other OMG publications, contact the Object Management Group, Inc. at:

OMG Headquarters
250 First Avenue
Needham, MA 02494
USA
Tel: +1-781-444-0404
Fax: +1-781-444-0320
pubs@omg.org
http://www.omg.org

Acknowledgments

The following companies submitted and/or supported parts of this specification:

- Borland Software Corporation
- Highlander Engineering, Inc.
- Nokia
- Sonera Corporation
- University of Helsinki
- Vertel Corporation

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. The submitters have rejected all solutions which 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 RFP requested a quite comprehensive specification to cover wireless access, terminal mobility and service provisioning in a mobile environment. The submitters decided to take a minimalistic approach. The response has been 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.

1.3 References

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

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

Architectural Framework 2

Contents

This chapter contains the following sections.

Section Title	Page
"Key Concepts"	2-1
"Overall Architecture"	2-2

2.1 Key Concepts

The key concepts in the this specification are:

- Mobile IOR,
- Home Location Agent,
- Access Bridge,
- Terminal Bridge, and
- GIOP Tunneling Protocol.

The Mobile IOR is a relocatable object reference. It identifies the Access Bridge and the terminal on which the target object resides. In addition, it identifies the Home Location Agent that keeps track of the Access Bridge to which the terminal is currently attached.

The Home Location Agent keeps track of the current location of the terminal. It provides operations to query and update terminal location. The Home Location Agent also provides operations to get a list of initial services and to resolve initial references in the home domain.

The Access Bridge is the network side end-point of the GIOP tunnel. It encapsulates the GIOP messages to the Terminal Bridge and decapsulates the GIOP messages from the Terminal Bridge. The Access Bridge also provides operations to get a list of initial services and to resolve initial references in the visited domain. The Access Bridge may also provide notifications of terminal mobility events.

The Terminal Bridge is the terminal side end-point of the GIOP tunnel. It encapsulates the GIOP messages to the Access Bridge and decapsulates the GIOP messages from the Access Bridge. The Terminal Bridge may also provide a mobility event channel that delivers notifications related to handoffs and connectivity losses.

The GIOP tunnel is the means to transmit GIOP messages between the Terminal Bridge and the Access Bridge. The generic GIOP Tunneling Protocol defines how GIOP messages are transmitted. The protocol also specifies necessary control messages to establish, release, and re-establish a GIOP tunnel. The proposed GIOP Tunneling Protocol (GTP) is an abstract, transport-independent protocol. This response defines three concrete tunneling protocols, that is the way how GTP messages are transmitted over TCP, UDP and WAP WDP.

2.2 Overall Architecture

The overall architecture is depicted in Figure 2-1. It identifies three different domains: home domain, visited domain, and terminal domain. The Home Domain for a given terminal is the domain that hosts the Home Location Agent of the terminal. A Visited Domain is a domain that hosts one or more Access Bridges through which it provides ORB access to some mobile terminals. The Terminal Domain consists of a terminal device that hosts an ORB and a Terminal Bridge through which the objects on the terminal can communicate with objects in other networks.

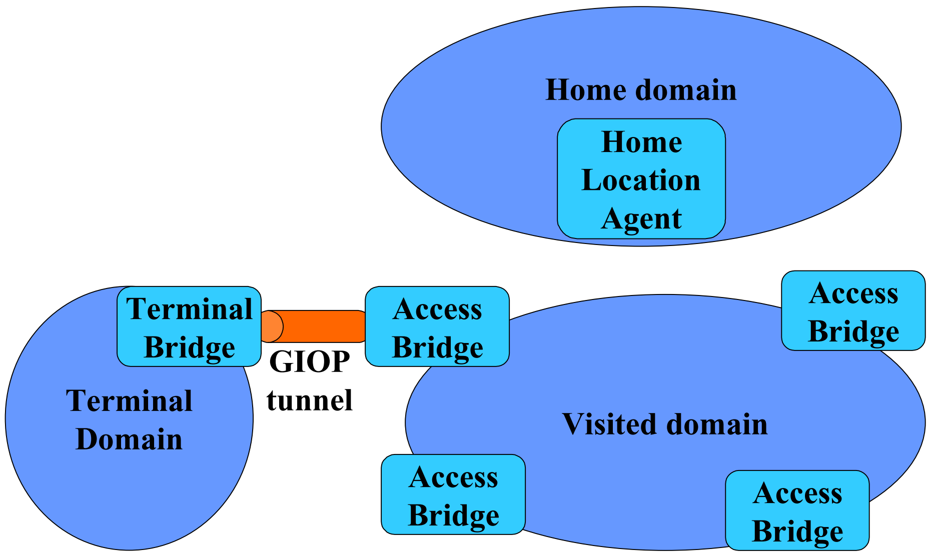

Figure 2-1 Architecture for Terminal Mobility in CORBA

Mobile IOR 3

Contents

This chapter contains the following sections.

Section Title	Page
"IIOP Profiles in Mobile IOR"	3-1
"The Mobile Terminal Profile"	3-2
"Translation to Mobile Target Object"	3-4
"Interoperability with GIOP 1.0 and 1.1"	3-5
"Additional Type Definitions"	3-5

A Mobile IOR is a special Interoperable Object Reference that hides the mobility of a terminal from clients that invoke operations on target objects located on the terminal. The Mobile IOR provides mobility transparency in a way that is itself transparent to the ORB that a client runs on. Hence the ORB that a non-mobile client runs on is not required to implement the Wireless Access and Terminal Mobility specification for terminal mobility to be available.

A Mobile IOR contains the normal IIOP Profile (**TAG_INTERNET_IOP**) required in an IOR, plus a 'Mobile Terminal' Profile (**TAG_MOBILE_TERMINAL_IOP**). There may be more than one IIOP Profile in the Mobile IOR. There can be only one Mobile Terminal Profile instance in the Mobile IOR.

3.1 IIOP Profiles in Mobile IOR

The ORB that a client runs on uses an IIOP Profile from the Mobile IOR (rather than the Mobile Terminal Profile) to route the client's invocations to the Access Bridge currently serving the terminal on which the target object is located.

The IIOP Profile or Profiles in a Mobile IOR have the normal structure defined in **IIOP::ProfileBody**, but they have additional semantics regarding the address and object key fields within that structure. These semantics are transparent to the client ORB that makes use of one of these Profiles.

3.1.1 Address information in IIOP Profiles in Mobile IORs

Instead of indicating the address of the target object, the host and port information in an IIOP Profile in a Mobile IOR indicate the address of either the target object's terminal's Home Location Agent or the Access Bridge that the terminal was last known to be associated with. When a Mobile IOR is created at the terminal, the terminal ORB chooses whether the address of the terminal's HLA or the Access Bridge the terminal is currently associated with is given in the IIOP Profile.

If the address in the IIOP Profile is that of the terminal's Home Location Agent, rather than its last known Access Bridge, when a client first performs an invocation upon the Mobile IOR, the HLA replies with a GIOP **LOCATION_FORWARD** message returning the Mobile IOR indicating the Access Bridge that the HLA believes the terminal is currently associated with.

If the address in the IIOP Profile is that of an Access Bridge rather than an HLA, the terminal may no longer be associated with that Access Bridge when a client makes its first invocation upon the Mobile IOR. If the terminal is now associated with another Access Bridge, the contacted Access Bridge should reply with a GIOP **LOCATION_FORWARD** message returning the Mobile IOR indicating the Access Bridge that the terminal is currently associated with.

Similarly, if at any time after a client has made its first invocation upon a Mobile IOR the terminal becomes associated with another Access Bridge, then the contacted Access Bridge should reply to the client's next invocation with a GIOP **LOCATION_FORWARD** message returning the Mobile IOR indicating the Access Bridge that the terminal is now associated with.

3.1.2 Mobile Object Key Format

In order to allow clients to make invocations from ORBs that only support versions of GIOP prior to version 1.2, Mobile IORs may optionally use a special format for the contents of the object key field within their IIOP Profiles. For details of this format see Section 3.4, "Interoperability with GIOP 1.0 and 1.1," on page 3-5.

3.2 The Mobile Terminal Profile

The Mobile Terminal Profile within a Mobile IOR contains information that the Home Location Agent and Access Bridges require to provide mobility transparency for target objects that have Mobile IORs. The information is not required by the ORB that a client of a Mobile IOR runs on, and hence only ORBs used to implement Home Location Agents and Access Bridges need to be able to use this profile type.

3.2.1 Mobile Terminal Profile Structure

A Mobile Terminal profile is an **IOP::TaggedProfile** with a tag value of **TAG_MOBILE_TERMINAL_IOP** and profile data with the structure defined by MobileTerminal::ProfileBody.

```
    const IOP::ProfileID        TAG_MOBILE_TERMINAL_IOP = ????;
```

Note – The constant value is to be assigned by the OMG and added to the IOP module in the CORBA specification.

```
module MobileTerminal {

    typedef sequence<octet> TerminalId;
    typedef sequence<octet> TerminalObjectKey;

    struct Version {
        octet     major;
        octet     minor;
    };

    struct ProfileBody {
        Version                   mior_version; // version of Mobile IOR
        octet                     reserved;
        TerminalId                terminal_id; // unique terminal identifier
        TerminalObjectKey         terminal_object_key;// object_key on terminal
        sequence <IOP::TaggedComponent> components;
    };

            ...

};
```

The **MobileTerminal::ProfileBody** structure identifies the version of the Mobile Terminal Profile (in **mior_version** element) used, the id of the terminal the target object resides on and the object key of the target object on the terminal. It may optionally include one or more tagged components. A **TAG_HOME_LOCATION_INFO** component is specified, and may be present in the Mobile Terminal Profile's component list. See Section 3.2.2, "TAG_HOME_LOCATION_INFO Component," on page 3-4.

There can be only one Mobile Terminal profile instance in the Mobile IOR.

The version of the Mobile Terminal Profile defined in this specification is 1.0 (major 1, minor 0).

3.2.2 TAG_HOME_LOCATION_INFO Component

The **TAG_HOME_LOCATION_INFO** component identifies the Home Location Agent of the terminal on which the Mobile IOR was created. If the mobile terminal has an Home Location Agent, then the **TAG_HOME_LOCATION_INFO** component must be present in the Mobile Terminal Profile.

If the mobile terminal does not have an Home Location Agent, then the object reference is only valid as long as the current GIOP tunnel between the Terminal Bridge and the Access Bridge exists. Such a terminal is referred as a "homeless terminal" in this specification.

The **TAG_HOME_LOCATION_INFO** component has a Home Location Agent object reference as its associated value, encoded as the CDR encapsulation of the data structure **MobileTerminal::HomeLocationInfo**.

```
    const IOP::ComponentID      TAG_HOME_LOCATION_INFO = ????;
```

Note – The constant value is to be assigned by the OMG and go into the IOP module in the CORBA Core.

```
module MobileTerminal {

            ...

    struct HomeLocationInfo {
        HomeLocationAgent       agent;
    };

            ...

};
```

The **TAG_HOME_LOCATION_INFO** component can appear at most once in a **TAG_MOBILE_TERMINAL_IOP** profile.

3.3 Translation to Mobile Target Object

The first time a Home Location Agent or Access Bridge receives a GIOP message for an invocation on a particular Mobile IOR it needs some way to establish the terminal id and object key of the mobile target object, and associate it with the object key included in the GIOP message (so that in the future it will know that messages containing that object key are intended for that same mobile target object.)

In GIOP 1.2 the Home Location Agent or Access Bridge can reply to the first message with the status **NEEDS_ADDRESSING_MODE**, to request the object reference of the target object. It can then examine the contents of the Mobile Terminal profile within that object reference, to obtain the terminal id and object key. However, that solution excludes clients running on an ORB using GIOP 1.0 or 1.1 from invoking on the Mobile IOR, as the **NEEDS_ADDRESSING_MODE** status cannot be returned to them by the HLA or Access Bridge.

3.4 Interoperability with GIOP 1.0 and 1.1

Since, in GIOP 1.0 and 1.1 the object key is the only available way of identifying the target from data in a GIOP Request header, a special Mobile Object Key (MOK) format is specified to allow invocations from GIOP 1.0 and 1.1 clients to be made on mobile target objects. It is a structure that may optionally be used to format the contents of the object key in the IIOP profile in the Mobile IOR.

When the MOK format is used, the contents of the object key is an encapsulation of four octets with the ASCII values ‘M’, ‘I’, ‘O’, ‘R’ followed by the structure **MobileTerminal::MobileObjectKey**.

```
module MobileTerminal {
    ...
    struct MobileObjectKey {
        Version mior_version;
        octet reserved;
        TerminalId terminal_id;
        TerminalObjectKey terminal_object_key;
    };
};
```

Use of the MOK format is optional. Even when the MOK format is used, the Mobile Terminal Profile is still included in the Mobile IOR - which means the terminal id and target object information are included twice in the object reference. This redundancy is allowed because the MOK solution is only offered to support legacy ORBs that do not support GIOP 1.2. The GIOP 1.2 mechanism is preferred, and hence always supported to assist the migration of systems to GIOP 1.2 support.

If the MOK format is used, the contents of the formatted key are only examined by the Home Location Agent and Access Bridge, which will use ORBs that implement the this specification. The MOK is not examined by client ORBs, which continue to consider the object key as an opaque piece of data. Hence non-mobile aware client ORBs are able to interoperate with target objects which have Mobile IORs that use the MOK format.

3.5 Additional Type Definitions

The **MobileTerminal** module contains all the type definitions used in this specification. They are provided below.

```
module MobileTerminal {

        ...

    typedef sequence<octet>        GIOPEncapsulation; // used in GIOP tunneling
    typedef sequence<octet>          GTPEncapsulation; // used in GTP forwarding

    enum HandoffStatus {
          HANDOFF_SUCCESS,
          HANDOFF_FAILURE,
          NO_MAKE_BEFORE_BREAK
    };                // used to report status of handoff

    struct GTPInfo {
          Version          gtp_version; // version of the GTP
          octet            protocol_level; // identifies GIOP Tunneling Protocol Level
          octet            protocol_id; // identifies GIOP Tunneling Protocol
    };                // identifies the GIOP Tunneling Protocol
    // values 0xE0...0xFF of protocol_id element arereserved for internal use

    const octet          TCP_TUNNELING = 0;
    const octet          UDP_TUNNELING = 1;
    const octet          WAP_TUNNELING = 2;

    struct AccessBridgeTransportAddress {
          GTPInfo                   tunneling_protocol;
          sequence<octet>           transport_address;
    };          // identifies transport access point of the Access Bridge

       typedef sequence<AccessBridgeTransportAddress> AccessBridgeTransportAddress-
    List;

    typedef string ObjectId; // same as CORBA::ORB::ObjectId
    typedef sequence<ObjectId> ObjectIdList
    // same as CORBA::ORB::ObjectIdList

};
```

Home Location Agent 4

Contents

This chapter contains the following sections.

Section Title	**Page**
"Location Update"	4-1
"Discovery"	4-4
"Message Processing"	4-4
"Terminal Ids"	4-5

The Home Location Agent keeps track of the Access Bridge that a mobile terminal is currently associated with. That is, which Access Bridge objects on the terminal can currently be invoked. It provides operations to update and to query the current location. It also provides operations resolve initial references in the Home Domain.

4.1 Location Update

The **HomeLocationAgent** interface provides operations for Access Bridges to carry out location updates and to query the current location of a terminal. The terminal is identified by a terminal identifier, **terminal_id**. The Home Location Agent may require the use of the CORBA Security Service to invoke the **update_location, deregister_terminal**, and **query_location** operations.

```
module MobileTerminal {

    interface HomeLocationAgent {

        void update_location (
            in TerminalId terminal_id,
            in AccessBridge new_access_bridge
        ) raises (UnknownTerminalId, IllegalTargetBridge);

        ...
    };
};
```

Parameters

terminal_id	Terminal for which the location update is done.
new_access_bridge	Object reference of the Access Bridge that wants to serve the terminal.

Exceptions

UnknownTerminalId	The HLA raises this exception, if it is not the HLA serving the terminal identified by the **terminal_id**.
IllegalTargetBridge	The HLA raises this exception, if it does not accept the Access Bridge identified by **new_access_bridge** to serve the terminal identified by **terminal_id**.

```
module MobileTerminal {

    interface HomeLocationAgent {

                    ...

            boolean deregister_terminal (
                in TerminalId terminal_id,
                in AccessBridge old_access_bridge
            ) raises (UnknownTerminalId);

            ...
        };
};
```

Return values

true	implies that the HLA believes that **old_access_bridge** manages the treminal.
false	implies that the HLA has already received an **update_location** operation from another Access Bridge.

Parameters

terminal_id	Terminal that has disappeared
old_access_bridge	Object reference of the Access Bridge that has lost the terminal

Exceptions

UnknownTerminalId	The HLA raises this exception, if it is not the HLA serving the terminal identified by the **terminal_id**.

When an Access Bridge has lost the terminal, it deregisters the location of the terminal by invoking **deregister_terminal**(**terminal_id**, **access_bridge_reference**) at the Home Location Agent.

If **deregister_terminal** operation returns **false**, the Access Bridge should keep the state information of the terminal since most probably the access recovery process is going on.

If an Access Bridge needs to query the current location of a terminal, that is the Access Bridge currently serving the terminal, it can invoke the **query_location** operation at the Home Location Agent of the terminal. The Home Location Agent may require the use of the CORBA Security Service to invoke the **query_location** operation.

```
module MobileTerminal {

    interface HomeLocationAgent {

        ...

    void query_location (
        in TerminalId terminal_id,
        out AccessBridge current_access_bridge
    ) raises (UnknownTerminalId, UnknownTerminalLocation);

        ...

    };
};
```

Parameters

terminal_id	Identifies the terminal the location of which is queried.
current_access_bridge	Object reference of the Access Bridge to which the HLA believes that the terminal is currently attached.

Exceptions

UnknownTerminalId	The HLA raises this exception, if it is not the HLA serving the terminal identified by the **terminal_id**.
UnknownTerminalLocation	The HLA raises this exception, if the given terminal has not registered its current location through an Access Bridge, or the paging procedure did not find the Access Bridge to which the terminal is attached.

4.2 Discovery

The Home Location Agent provides discovery operations so that the terminals can resolve initial references to CORBA services available in the Home Domain. The operations are **list_initial_services** and **resolve_initial_references**. They are the same as provided by the ORB pseudo interface for local applications.

```
module MobileTerminal {

    interface HomeLocationAgent {

        ...

        ObjectIdList list_initial_services();
        Object resolve_initial_references(
            in ObjectId identifier
        ) raises(InvalidName);

    };
};
```

4.3 Message Processing

When the Home Location Agent receives a GIOP message targeted to a terminal, its behavior depends on whether or not it currently has an Access Bridge associated with that terminal. If it does, it replies with the **LOCATION_FORWARD** status and returns the Mobile IOR identifying the current Access Bridge. If not, it replies with the system exception OBJECT_NOT_EXIST (to a Request) or with the UNKNOWN_OBJECT status (to a Locate Request).

4.4 Terminal Ids

The **TerminalId**s need to be unique world-wide.

One possible scheme that may be used to achieve this is to concatenate the following information to produce each identifier:

- IP version (1 byte)
- IP address (4 or 16 bytes), and
- **local_id** (variable number of bytes).

The IP address can be any IP address that is owned by the organization that is generating the TerminalId, and the **local_id** is a unique identifier within that organization.

Any other scheme may be used though, as long as it produces globally unique identifiers.

Access Bridge 5

Contents

This chapter contains the following sections.

Section Title	**Page**
"Discovery"	5-1
"Query"	5-2
"Message Processing"	5-2
"Mobility Event Notifications"	5-3

The Access Bridge encapsulates/decapsulates the GIOP messages to/from the Terminal Bridge using a GIOP Tunneling Protocol. It also provides operations to get a list of initial services and to resolve initial references in the visited domain. In addition, the Access Bridge may support handoff. The Access Bridge may also provide notifications related to movements of terminals.

GIOP Tunneling Protocol is described in Chapter 7. The handoff procedures are described in Chapter 8.

5.1 Discovery

The Access Bridge provides discovery operations so that the terminals can resolve initial references to CORBA services available in the Visited Domain. The operations are **list_initial_services** and **resolve_initial_references**. They are the same as provided by the ORB **pseudo** interface for local applications.

```
module MobileTerminal {

    interface AccessBridge {
        ObjectIdList list_initial_services();
        Object resolve_initial_references(
            in ObjectId identifier
        ) raises(InvalidName);

        ...

    };
};
```

5.2 Query

The Access Bridge also provides query operations that can be used to query whether or not a specific terminal is attached to the bridge, and the address information for the Access Bridge:

```
module MobileTerminal {

    interface AccessBridge {

        ...

        Boolean terminal_attached (
            in TerminalId terminal_id
        );

        void get_address_info (
            out AccessBridgeTransportAddessList transport_address_list
        );

        ...

    };
};
```

If the HLA requires the CORBA Security Service to be used in location update, then the Access Bridge must use the CORBA Security Service to protect the usage of the terminal_attached operation. The Access Bridge may also use the CORBA Security Service to protect the **get_address_info** operation.

5.3 Message Processing

The Access Bridge acts as a relay between the server and client. It maintains bindings between **terminal_id** and the transport address of the GIOP tunnel to the terminal. For each terminal the Access Bridge keeps a state of outstanding invocations. An outstanding invocation is a GIOP message to which a reply is expected.

When the bridge gets a message targeted to a terminal, it encapsulates the message to the GIOP tunneling protocol in use and sends it to the GIOP tunnel address associated with the **terminal_id**.

If the Access Bridge does not have a tunneling association with the terminal, then it can query the current location of the terminal from the HLA or it can replace the IOR so that the HLA is in the IIOP Profile. In both cases the Access Bridge must reply with the **LOCATION_FORWARD** status.

If the IOR does not have **TAG_HOME_LOCATION_INFO** component or the Access Bridge does not know the HLA of the terminal, then the Access Bridge must reply with the system exception OBJECT_NOT_EXIST to a Request and with the UNKNOWN_OBJECT status to a Locate Request.

If the Access Bridge gets a reply the target of which is on a terminal that has moved to a new Access Bridge, it can use the forwarding mechanism described in Chapter 8. If the Access Bridge does not support handoff, then it should silently discard the Reply message.

When the Access Bridge gets an encapsulated GIOP message from a terminal, it decapsulates the message and forwards it to the target.

5.4 Mobility Event Notifications

The Access Bridge may, optionally, raise terminal mobility related events through a Notification Service Event Channel. The following Event types are defined so that if the Access Bridge does this, it may use standard events:

```
module MobileTerminalNotification {

    struct HandoffDepartureEvent {
        MobileTerminal::TerminalId terminal_id;
        MobileTerminal::AccessBridge new_access_bridge;
    };

    struct HandoffArrivalEvent {
        MobileTerminal::TerminalId terminal_id;
        MobileTerminal::AccessBridge old_access_bridge;
    };

    struct AccessDropoutEvent {
        MobileTerminal::TerminalId terminal_id;
    };

    struct AccessRecoveryEvent {
        MobileTerminal::TerminalId terminal_id;
    };

    ...

};
```

When a terminal moves from an old Access Bridge to a new Access Bridge, the old Access Bridge supplies the **HandoffDepartureEvent** and the new Access Bridge supplies the **HandoffArrivalEvent**.

When a terminal establishes the GIOP tunnel to the Access Bridge for the first time, then Handoff, then the the new Access Bridge supplies the **HandoffArrivalEvent** with NIL as reference to the old Access Bridge. When a terminal closes the GIOP tunnel to the Access Bridge, then the Access Bridge supplies the **HandoffDepartureEvent** with NIL as reference to the new Access Bridge.

When an Access Bridge detects that transport connectivity to a terminal has dropped, it supplies the **AccessDropoutEvent**. If the terminal re-establishes the GIOP Tunnel to the same Access Bridge, then the Access Bridge supplies the **AccessRecoveryEvent** if it has supplied the **AccessDropoutEvent**. If the terminal re-establish the GIOP Tunnel to a new Access Bridge, then the old Access Bridge supplies the **HandoffDepartureEvent** and the new Access Bridge supplies the **HandoffArrivalEvent**.

Terminal Bridge 6

Contents

This chapter contains the following sections.

Section Title	**Page**
"Mobility Event Notifications"	6-1

The Terminal Bridge encapsulates/decapsulates the GIOP messages to/from the Access Bridge using a GIOP Tunneling Protocol. The Terminal Bridge may support handoff. As an optional feature, the Terminal Bridge may also provide notifications of mobility related events for mobility-aware applications on the mobile terminal.

GIOP Tunneling Protocol and handoff procedures are described in Chapters 7 and 8, respectively.

6.1 *Mobility Event Notifications*

The Terminal Bridge may, optionally, raise terminal mobility related events through a Notification Service Event Channel. The following Event types are defined so that if the Terminal Bridge does this, it may use standard events.

```
module TerminalMobilityNotification {

    ...

    struct TerminalHandoffEvent {
        MobileTerminal::AccessBridge new_access_bridge;
    };

    struct TerminalDropoutEvent {
        MobileTerminal::TerminalId terminal_id;
    };

    struct TerminalRecoveryEvent {
        MobileTerminal::TerminalId terminal_id;
    };
};
```

When the Terminal Bridge detects that it has lost transport connectivity to the Access Bridge, it supplies the **TerminalDropoutEvent**. When the GIOP Tunnel has been re-established, then the Terminal Bridge generates the **TerminalRecoveryEvent** if the Access Bridge is the same as before. If the Access Bridge is different, then the Terminal Bridge supplies the **TerminalHandoffEvent**.

When a handoff takes place, the Terminal Bridge supplies the **TerminalHandoffEvent**. The Terminal Bridge also supplies the **TerminalHandoffEvent**, when the Terminal establishes the GIOP Tunnel to an Access Bridge for the first time. When the Terminal Bridge closes the GIOP Tunnel, then it supplies the **TerminalHandoffEvent** with NIL as the **new_access_bridge**.

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”	7-23

A GIOP tunnel is the means to transmit GIOP and tunnel control messages between a Terminal Bridge and an Access Bridge. There is only ever 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.

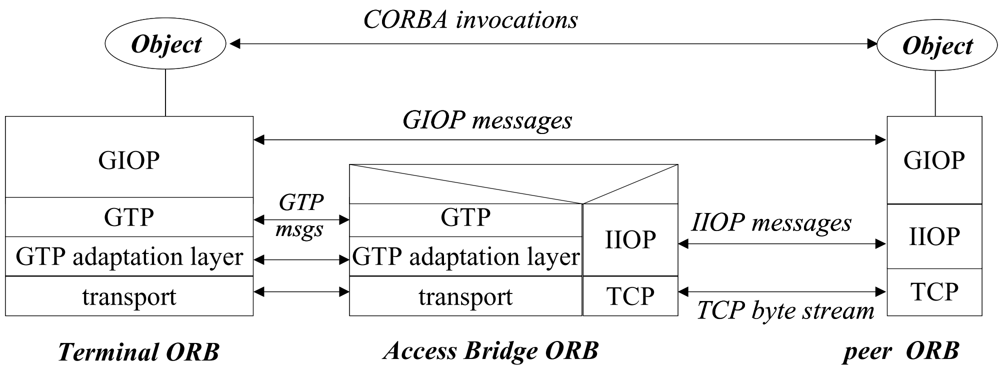

Figure 7-1 GIOP Tunneling Protocol Architecture

Since the GIOP Tunneling Protocol is an abstract protocol, it needs to be mapped onto one or more concrete protocols. This specification defines three concrete tunneling protocols: TCP Tunneling, UDP Tunneling, and WAP 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 either the highest sequence number of GTP messages received by the sender or the highest sequence number of GIOPData message succesfully received and processed by the sender (see Special Notes in Section 7.2.15).

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

Table 7-1 GTP Messages

Message name	gtp_msg_type	GTP Level
ReleaseTunnelRequest	0x03	1, 2
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): **RECOVERY_REQUEST**,

- Network Initiated Handoff (see Section 8.2): **NETWORK_REQUEST**,
- Terminal Initiated Recovery (see Section 8.3): **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 word, 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 establish 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. In order to avoid allocation conflicts, Access Bridges uses even numbers and Terminal Bridge uses odd numbers (but not 0xFFFFFFFF, which is reserved as an error indicator; see Section 7.2.11). 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 a **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**. In order to avoid allocation conflicts, Access Bridges uses even numbers and Terminal Bridge uses 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 as 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 recepient
    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 implicilty indicated by normal acknowledgement 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. An **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 it's **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

FORWAD_ERROR_UNKNOWN_SENDER. The Access Bridge will then consider that terminal lost, and begin tearing down it's 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 starts 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 has 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.

7.4.2 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

7.4.3 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.

7.4.4 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.

7.4.5 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.

7.4.6 Resume

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

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.

7.4.7 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 has been received.

Both Access and Terminal Bridge can use this chunk.

7.4.8 GTPData

The chunk Type is 0x06. The Flags field (one octet) indicate 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.

7.5 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 no 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 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 number 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].

Handoff and Access Recovery 8

Contents

This chapter contains the following sections.

Section Title	**Page**
"Initiation"	8-2
"Network Initiated Handoff"	8-3
"Terminal Initiated Handoff"	8-8
"Access Recovery"	8-11
"GTP Message Forwarding"	8-14
"Terminal Tracking"	8-17

Generally, a handoff consists of three distinct phases: the information gathering phase, the decision phase, and the execution phase. Bridge handoff, that is the handoff which is visible on the ORB level, is a part of the execution phase in cases where the mobile terminal moves from one Access Bridge to another.

The handoff support is an optional feature of this specification. The level of the GIOP Tunneling Protocol identifies whether (Level 2) or not (Level 1) handoff support is available.

There are two different cases of handoff:

1. backward handoff
2. forward handoff (access recovery).

The first one is the normal case whereas the second one is performed in order to re-establish connectivity after a sudden loss. In the following we use the term *handoff* to mean the backward handoff and the term *access recovery* to mean the forward handoff.

The handoff may be *network initiated* or *terminal initiated*. The access recovery is always terminal initiated.

8.1 Initiation

The **AccessBridge** interface contains the **start_handoff** operation, which is called by an external handoff control application to initiate the handoff procedure. In the **MobileTerminal** module there is also the **HandoffCallback** interface that contains the **report_handoff_status** operation, which is used by the Access Bridge to report the outcome status of handoff to the external handoff control application.

```
module MobileTerminal {

      ...

    interface HandoffCallback {
        void report_handoff_status (
            in HandoffStatus status
        );
    };

      ...

};
```

Parameters

status	Outcome status of handoff procedure.

```
module MobileTerminal {

      ...

    interface AccessBridge {

      ...

        void start_handoff(
            in TerminalId terminal_id,
            in AccessBridge new_access_bridge,
            in HandoffCallback handoff_callback_target
        );
        ...
    };
```

```
        ...
};
```

Parameters

terminal_id	Identifies the terminal to be moved to a new Access Bridge.
new_access_bridge	Reference to the new Access Bridge.
handoff_callback_target	Object to which the status of handoff will be reported.

8.2 Network Initiated Handoff

The network initiated handoff starts when an external application invokes the **start_handoff** operation in the Access Bridge currently serving the terminal. In the description below this Access Bridge is referred to as the old Access Bridge. The Access Bridge to which the terminal moves is referred to as the new Access Bridge.

The handoff procedure assumes that the terminal can establish connectivity to the new Access Bridge before releasing the connectivity to the old Access Bridge. If this cannot be done, then the alternative procedure that is described in Section 8.2.5, "Alternative Handoff Procedure," on page 8-6 must be used.

8.2.1 Old Access Bridge

1. The old Access Bridge gets involved when the **start_handoff** operation is invoked on it.
2. The old Access Bridge invokes the **transport_address_request** operation in the new Access Bridge, which returns a list of transport addresses of the new Access Bridge and a Boolean value indicating whether or not the new Access Bridge accepts the terminal.
3. If the terminal is not accepted, then the old Access Bridge only reports the **HANDOFF_FAILURE** status by invoking the **report_handoff_status** operation at the **handoff_callback_target** and the handoff procedure is (unsuccessfully) completed. The old Access Bridge continues to serve the Terminal Bridge as the current Access Bridge.
4. If the terminal was accepted by the new Access Bridge, then the old Access Bridge sends the **HandoffTunnelRequest** message to the Terminal Bridge.
5. The following two steps (6 and 7) can take place in any order.
6. When the old Access Bridge gets the **HandoffTunnelReply** message from the Terminal Bridge, then

- if the status indicates a failure in handoff, then the old Access Bridge reports the **HANDOFF_FAILURE** status by invoking the **report_handoff_status** operation at the **handoff_callback_target** and the handoff procedure is (unsuccessfully) completed. The old Access Bridge continues to serve the Terminal Bridge as the current Access Bridge.
- if the status is **NO_MAKE_BEFORE_BREAK**, then the old Access Bridge waits for **recovery_request** before reporting the handoff status using the **report_handoff_status** operation at the **handoff_callback_target**. (See also section 8.2.5.)
- if the status indicates a successful handoff, then the old Access Bridge waits for the **ReleaseTunnelRequest** message from the Terminal Bridge. After that it send the **ReleaseTunnelReply** message to the Terminal Bridge and releases its transport end-point to the Terminal Bridge.

7. When the new Access Bridge invokes the **handoff_completed** operation at the old Access Bridge, then the old Access Bridge knows that the new Access Bridge has taken the responsibility of the terminal.

8. It is assumed that the handoff status received by the old Access Bridge from the Terminal Bridge and the new Access Bridge is same. If they are not the same, then the old Access Bridge takes implementation depended actions to recover this error situation.

9. The old Access Bridge notifies all other Access Bridges interested in movements of the terminal (see Section 8.6, “Terminal Tracking,” on page 8-17).

10. If the old Access Bridge supports Mobility Event Notifications, it generates a notification of a departing terminal.

11. The old Access Bridge reports the handoff status by invoking the **report_handoff_status** operation at the **handoff_callback_target**.

8.2.2 New Access Bridge

1. The new Access Bridge gets involved when the old Access Bridge invokes the **transport_address_request** operation at the new Access Bridge. If the new Access Bridge does not accept the terminal, then nothing needs to be done. The new Access Bridge should take the invocation of the **transport_address_request** operation only as a hint of a forthcoming handoff because the Terminal Bridge may use the access recovery procedure instead of the handoff procedure; see Section 8.2.5, “Alternative Handoff Procedure,” on page 8-6.

2. The new Access Bridge gets the **EstablishTunnelRequest** message from the Terminal Bridge.

3. The new Access Bridge invokes the **update_location** operation at the Home Location Agent.

4. The new Access Bridge sends the **EstablishTunnelReply** message to the Terminal Bridge.

5. The new Access Bridge invokes the **handoff_completed** operation at the old Access Bridge.

6. If the location update failed, then the new Access Bridge frees its transport end-point to the Terminal Bridge.

7. If the location update was successful and the new Access Bridge supports Mobility Event Notifications, it generates a notification of an arriving terminal.

8.2.3 Terminal Bridge

1. The Terminal Bridge gets involved when it receives the **HandoffTunnelRequest** message from the old Access Bridge.

2. The Terminal Bridge establishes transport connectivity to the new Access Bridge. If this fails, then the Terminal Bridge sends the **HandoffTunnelReply** message to the old Access Bridge that indicates a handoff failure, and the handoff procedure is (unsuccessfully) completed. The Terminal Bridge continues to use the GIOP Tunnel to the old Access Bridge.

3. The Terminal Bridge sends the **EstablishTunnelRequest** message to the new Access Bridge,

4. The Terminal Bridge waits for the **EstablishTunnelReply** message from the new Access Bridge.

5. The Terminal Bridge sends the **HandoffTunnelReply** message to the old Access Bridge.

6. If the request of tunnel establishment was rejected, then the Terminal Bridge continues to use the tunnel to the old Access Bridge.

7. If the tunnel to the new Access Bridge was granted, then the Terminal Bridge sends the **ReleaseTunnelRequest** message to the old Access Bridge. After receiving the **ReleaseTunnelReply** message from the old Access Bridge, the Terminal Bridge can release its transport end-point to the old Access Bridge.

8. If the Terminal Bridge supports Mobility Event Notifications, it generates a notification of handoff.

8.2.4 Message Sequence Chart

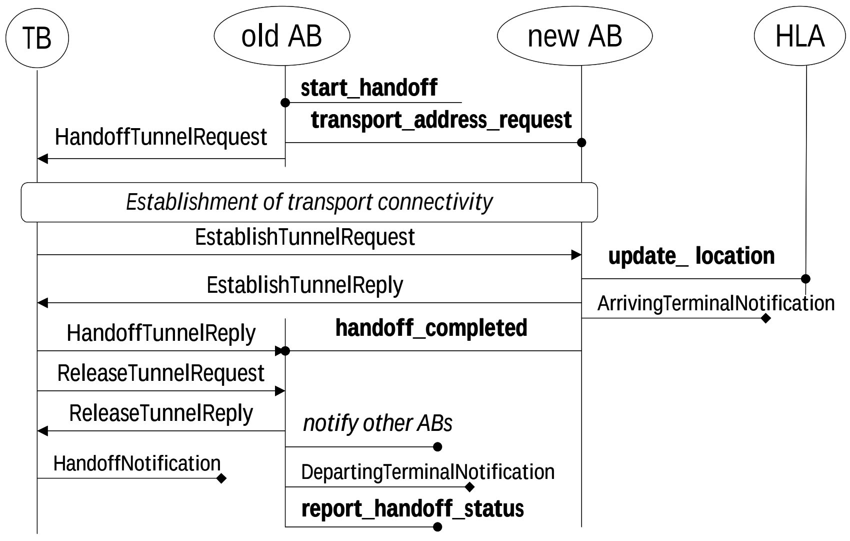

Figure 8-1 Message Sequence Chart

8.2.5 Alternative Handoff Procedure

If the terminal cannot have simultaneous transport connectivity to the old and new Access Bridge, then the following procedure is used by the Terminal Bridge.

1. The Terminal Bridge gets involved when it receives the **HandoffTunnelRequest** message from the old Access Bridge.
2. The Terminal Bridge sends the **HandoffTunnelReply** message to the old Access Bridge in which the handoff status NO_MAKE_BEFORE_BREAK.
3. The Terminal Bridge sends the **ReleaseTunnelRequest** message to the old Access Bridge and waits for the **ReleaseTunnelReply** from the old Access Bridge.
4. The Terminal Bridge releases its transport end-point to the old Access Bridge.
5. The Terminal Bridge establish GIOP Tunnel to the new Access Bridge using the access recovery procedure described in Section 8.4, “Access Recovery,” on page 8-11.

The old Access Bridge sees from the handoff status of NO_MAKE_BEFORE_BREAK that the terminal will use the access recovery procedure instead of the handoff procedure. The new Access Bridge sees this alternative handoff procedure as usual access recovery procedure.

When the old Access Bridge receives NO_MAKE_BEFORE_BREAK as response to the **HandoffTunnelReply** message, then the Access Bridge waits for **recovery_request** before reporting the handoff status using the **report_handoff_status** operation at the **handoff_callback_target**.

8.2.6 IDL

```
module MobileTerminal {

    ...

    interface AccessBridge {

    ...

    void transport_address_request(
            // Called by the old Access Bridge at the new Access Bridge
        in TerminalId terminal_id,
        out AccessBridgeTransportAddressList new_access_bridge_addresses,
        out boolean terminal_accepted
    );

    ...

    };

    ...

};
```

Parameters

terminal_id	Identification of terminal that will move.
new_access_bridge_addresses	List of transport addresses that the terminal can contact in order to establish transport connectivity.
terminal_accepted	FALSE, if the called Access Bridge does not accept the terminal.

```
module MobileTerminal {

    ...

    interface AccessBridge {

    ...

        void handoff_completed(
                // called by the new Access Bridge at the old Access Bridge
            in TerminalId terminal_id,
            in HandoffStatus status
        ) ;
        ...
    };
        ...
};
```

Parameters

terminal_id	Identifies the terminal.
status	Status of handoff.

8.3 Terminal Initiated Handoff

The terminal initiated handoff procedure requires that the terminal can establish connectivity to the new Access Bridge before releasing the connectivity to the old Access Bridge. If this cannot be done, then the terminal initiated handoff must be done using the access recovery mechanism: The Terminal Bridge closes connectivity to the old Access Bridge and then carries out the access recovery to the new Access Bridge.

Below we describe action taken by the Terminal Bridge and by the new and old Access Bridges.

8.3.1 Terminal Bridge

1. The Terminal Bridge establishes transport connectivity to the new Access Bridge.
2. The Terminal Bridge sends the **EstablishTunnelRequest** message to the new Access Bridge.
3. The Terminal Bridge waits for the **EstablishTunnelReply** message from the new Access Bridge.
4. If the tunnel establishment was rejected, then the Terminal Bridge releases its transport end-point to the new Access Bridge and the handoff procedure is (unsuccessfully) completed. The Terminal Bridge continues to use the GIOP Tunnel to the old Access Bridge.
5. The Terminal Bridge sends the **ReleaseTunnelReuqest** message to the old Access Bridge.
6. After receiving the **ReleaseTunnelReply** message from the old Access Bridge, the Terminal Bridge can release its transport end-point to the old Access Bridge.
7. If the Terminal Bridge supports Mobility Event Notifications, it generates a notification of handoff.

8.3.2 New Access Bridge

1. The new Access Bridge gets involved when it receives the **EstablishTunnelRequest** message from the Terminal Bridge.
2. The new Access Bridge invokes the **update_location** operation at the Home Location Agent.
3. If the location update failed, then the new Access Bridge sends the **EstablishTunnelReply** message to the Terminal Bridge and releases its transport end-point to the Terminal Bridge and the handoff procedure is (unsuccessfully) completed.
4. The new Access Bridge invokes the **handoff_in_progress** operation at the old Access Bridge.
5. The new Access Bridge sends the **EstablishTunnelReply** message to the Terminal Bridge.
6. If the new Access Bridge supports Mobility Event Notifications, it generates a notification of an arriving terminal.

8.3.3 Old Access Bridge

1. The old Access Bridge gets involved, when the new Access Bridges invokes the **handoff_in_progress** operation at the old Access Bridge.
2. The old Access Bridge waits for the **ReleaseTunnelRequest** message from the Terminal Bridge.
3. After sending the **ReleaseTunnelReply** message to the Terminal Bridge, the old Access Bridge can release its transport end-point to the Terminal Bridge.
4. The old Access Bridge notifies all other Access Bridges interested in movements of the terminal (see Section 8.6, "Terminal Tracking," on page 8-17).
5. If the old Access Bridge supports Mobility Event Notifications, it generates a notification of a departing terminal.

8.3.4 Message Sequence Chart

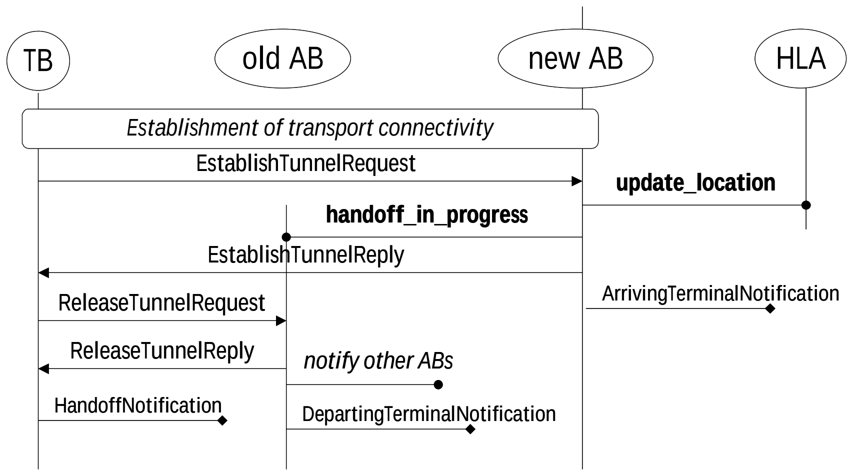

Figure 8-2 Message Sequence Chart

8.3.5 IDL

```
module MobileTerminal {

    ...

  interface AccessBridge {

    ...

      void handoff_in_progress (
              // called by the old Access Bridge in the new Access Bridge
         in TerminalId terminal_id,
         in AccessBridge new_access_bridge
      );

    ...

  };

    ...

};
```

Parameters

terminal_id	Identifies the terminal.
new_access_bridge	Reference of the new access bridge.

8.4 Access Recovery

When the Terminal Bridge detects that the connectivity to the Access Bridge is lost, a dropout notification is generate in the terminal domain and the Terminal Bridge starts the access recovery procedure. There are two possible successful outcomes of the access recovery procedure:

- The access is re-established to the same Access Bridge as before.
- The access is established to a new Access Bridge.

8.4.1 Recovery to the Old Access Bridge

8.4.1.1 Terminal Bridge

1. The Terminal Bridge establishes transport connectivity to an Access Bridge.
2. The Terminal Bridge sends the **EstablishTunnelRequest** message to the Access Bridge.
3. The Terminal Bridge waits for the **EstablishTunnelReply** message from the Access Bridge.
4. From the **EstablishTunnelReply** message the Terminal Bridge learns that the Access Bridge is the same as before and which is the last GTP message that the Access Bridge has received. The Terminal Bridge retransmits the lost GTP messages.
5. If the Terminal Bridge supports Mobility Event Notifications, it generates a recovery notification.

8.4.1.2 Old Access Bridge

1. The old Access Bridge receives the **EstablishTunnelRequest** from the Terminal Bridge.
2. From the **EstablishTunnelRequest** message the Access Bridge learns that the tunnel establishment is access recovery to it and which is the last GTP message that the Terminal Bridge has received.
3. The Access Bridge sends the **EstablishTunnelReply** message and retransmits the lost GTP messages.

4. If the old Access Bridge supports Mobility Event Notifications, it generates an access recovery notification only if it has generated the access dropout notification for the terminal.

8.4.2 Recovery to New Access Bridge

8.4.2.1 Terminal Bridge

1. same as in recovery to the old Access Bridge
2. same as in recovery to the old Access Bridge
3. same as in recovery to the old Access Bridge
4. From the **EstablishTunnelReply** message the Terminal Bridge learns that the Access Bridge is a new one and which is the last GTP message that the old Access Bridge has received. Another possibility is that the **EstablishTunnelReply** indicates location update failure, which terminates the recovery procedure.
5. If the Terminal Bridge supports the Mobility Event Notifications, then it generates a handoff notification.
6. The Terminal Bridge retransmits the GTP messages that the old Access Bridge has lost thru the new Access Bridge.

8.4.2.2 New Access Bridge

1. The new Access Bridge receives the **EstablishTunnelRequest** from the Terminal Bridge.
2. From the **EstablishTunnelRequest** message the Access Bridge learns that the tunnel establishment is access recovery to a new Access Bridge and which is the last GTP message that the Terminal Bridge has received.
3. The new Access Bridge invokes the **location_update** operation at the Home Location Agent.
4. If the location update fails, the new Access Bridge sends the **EstablishTunnelReply** message that indicates location update failure and completes the recovery procedure by releasing its transport end-point to the Terminal Bridge.
5. If the location update was successful, the new Access Bridge invokes the **recovery_request** operation at the old Access Bridge.
6. The new Access Bridge sends the **EstablishTunnelReply** message to the Terminal Bridge. If the **recovery_request** operation in step 5 raised an exception, then the new Access Bridge sends the **EstablishTunnelReply** message to the Terminal Bridge with status **ACCESS_REJECT_RECOVERY_FAILURE**.
7. If the new Access Bridge supports The Mobility Event Notifications, it generates a handoff arrival notification.

8. As long as needed the new Access Bridge forwards GTP messages between the Terminal Bridge and the old Access Bridge(s).

8.4.2.3 *Old Access Bridge*

1. The old Access Bridge gets involved when the new Access Bridge invokes the **recovery_request** operation at it. If the terminal is unknown, the old Access Bridge returns the **UnknownTerminalId** exception.
2. The old Access Bridge notifies other Access Bridges interested in movements of the terminal (see Section 8.6, "Terminal Tracking," on page 8-17).
3. If the old Access Bridge supports Mobility Event Notifications, it generates a notification of a departing terminal.
4. The old Access Bridge retransmits the GTP messages that the Terminal Bridge has lost through the new Access Bridge.

8.4.2.4 *Message Sequence Chart*

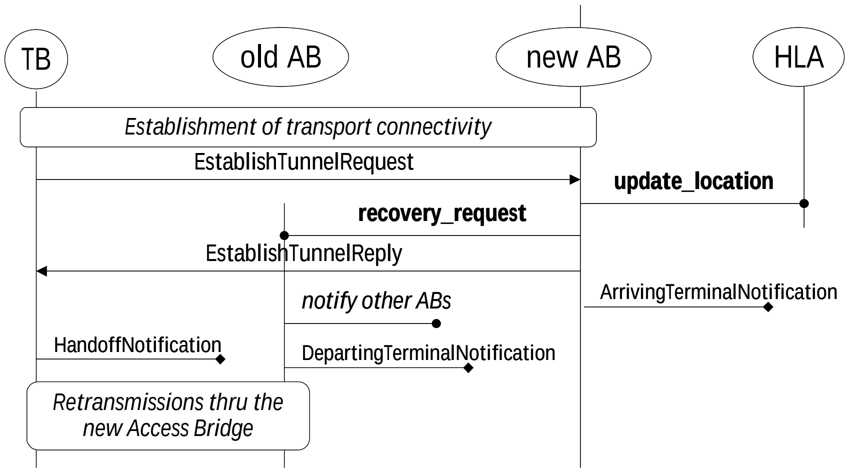
TB
old AB
new AB
HLA
Establishment of transport connectivity
EstablishTunnelRequest
update_location
recovery_request
EstablishTunnelReply
ArrivingTerminalNotification
notify other ABs
HandoffNotification
DepartingTerminalNotification
Retransmissions thru the new Access Bridge

Figure 8-3 Message Sequence Chart

8.4.2.5 *IDL*

```
module MobileTerminal {

    ...

  interface AccessBridge {

    ...

      void recovery_request (
              // called by the new Access Bridge in the old Access Bridge
          in TerminalId terminal_id,
          in AccessBridge new_access_bridge,
          in unsigned short highest_gtp_seqno_received_at_terminal,
          out unsigned short highest_gtp_seqno_received_at_access_bridge
      ) raises (UnknownTerminalId);

    ...

  };

    ...

};
```

Parameters

terminal_id	Identifies the terminal.
new_access_bridge	Reference of the new Access Bridge.
highest_gtp_stqno_received_at_terminal	Highest GTP sequence number that the Terminal Bridge has received from the Access Bridge.
highest_gtp_stqno_received_at_access_bridge	Highest GTP sequence number that the Access Bridge has received from the Terminal Bridge.

Expections

UnknownTerminalId	Indicates that the old Access Bridge does not (anymore) have the tunnel state for the terminal.

8.5 GTP Message Forwarding

The GIOP requires that replies are sent in the same GIOP connection as the request came in. Since an Access Bridge is the GIOP connection end-point, replies must go through it even if the terminal has moved to another Access Bridge. Therefore, the **AccessBridge** interface contains two operations to be used in relaying GTP messages between the Terminal Bridge and an old Access Bridge through the current Access Bridge.

When an old Access Bridge receives a GIOP message the actual destination of which is on a terminal that has moved, the old Access Bridge creates the corresponding GTP message(s) and invokes the **gtp_to_terminal** operation at the current Access Bridge. The old Access Bridge may use the **query_location** operation available in the

HomeLocationAgent interface to learn the current Access Bridge. The current Access Bridge uses the **GTPForward** message to deliver the GTP message to the Terminal Bridge.

When the Terminal Bridge wants to send a GIOP message thru an old Access Bridge, the Terminal Bridge creates the corresponding GTP message(s) and sends the **GTPForward** message(s) to the current Access Bridge. The current Access Bridge invokes the **gtp_from_terminal** operation at the old Access Bridge.

The old Access Bridge may response to the **gtp_from_terminal** with exception UnknownTerminalId. In this case, the current Access Bridge returns the **GTPForwardReply** message with status **FORWARD_ERROR_UNKNOWN_SENDER** to the Terminal Bridge.

When the current Access Bridge finds out the status of a forwarded GTP message received in a **gtp_to_terminal** invocation by an old Access Bridge, it invokes **gtp_acknowledge** at that Access Bridge, reporting the status of forwarding.

```
module MobileTerminal {

    ...

  interface AccessBridge {

    ...

      void gtp_to_terminal (
          in TerminalId terminal_id,
          in AccessBridge old_access_bridge,
          in unsigned long gtp_message_id,
          in GTPEncapsulation gtp_message
      ) raises (TerminalNotHere);

    ...

  };

    ...

};
```

Parameters

terminal_id	Identifies the terminal.
old_access_bridge	Identifies the Access Bridge from which the reply comes.
gtp_message_id	A handle used in a possible GTP reply message to identify to which GTP message the reply is.
gtp_message	Octet sequence containing the GTP message.

Exceptions

TerminalNotHere	Indicates that the terminal has moved from the invoked Access Bridge.

```
module MobileTerminal {

        ...

    interface AccessBridge {

        ...

            void gtp_from_terminal(
                in TerminalId terminal_id,
                in unsigned long gtp_message_id,
                in GTPEncapsulation gtp_message
            ) raises (UnknownTerminalId);

        ...

    };

        ...

};
```

Parameters

terminal_id	Identifies the terminal from which the GTP message is coming.
gtp_message_id	A handle to be used in a possible GTP reply message to identify to which GTP message the reply is.
gtp_message	Octet sequence containing the GTP message.

Exceptions

UnknownTerminalId	Indicates that the Access Bridge has already forgotten the identified terminal and can no longer accept forwarded messages from it.

```
module MobileTerminal {

    ...

  interface AccessBridge {

    ...

      void gtp_acknowledge (
          in unsigned long gtp_message_id,
          in unsigned long status
      );

    ...

  };

    ...

};
```

Parameters

gtp_message_id	The message id the caller received in the **gtp_to_terminal** call, to which this is an acknowledgement
status	The status of forwarding as received in a **GTPForwardReply** message (status field in GTPForwardReplyBody; see section 7.2.18)

8.6 Terminal Tracking

An Access Bridge needs to know the current Access Bridge of the terminal as long as the Terminal Bridge has open GIOP connections thru the Access Bridge. Therefore, the **AccessBridge** interface has two operations related to terminal tracking.

When a terminal moves from Access Bridge A to Access Bridge B, then Access Bridge A notifies the Access Bridge from which the terminal came (let it be Access Bridge C) and all other Access Bridges that have subscribed handoff notice of that terminal from the Access Bridge A (let they be Access Bridges D and E). If the Access Bridges C, D, and E still want to follow the terminal, they must subscribe the handoff notice from the Access Bridge B, that is to invoke the **subscribe_handoff_notice** operation at the Access Bridge B.

When the terminal moves from the Access Bridge B, the Access Bridge B notifies the Access Bridge A and those Access Bridges who has subscribed the notice. The operation is **handoff_notice**.

```
module MobileTerminal {

        ...

    interface AccessBridge {

        ...

        void handoff_notice (
            in TerminalId terminal_id,
            in AccessBridge new_access_bridge
        ) ;

        ...

    };

        ...

};
```

Parameters

terminal_id	Identifies the terminal that has just moved.
new_access_bridge	Reference to Access Bridge to which the terminal has moved.

```
module MobileTerminal {

    ...

    interface AccessBridge {

    ...

        void subscribe_handoff_notice (
            // called by an Access Bridge who wants to follow terminal movements
            in TerminalId terminal_id,
            in AccessBridge interested_access_bridge
        )  raises (TerminalNotHere);

    ...

    };

    ...

};
```

Parameters

terminal_id	Identifies the terminal to be followed.
interested_access_bridge	Reference to Access Bridge that wants to receive a handoff notice when the terminal moves again.

Exceptions

TerminalNotHere	Indicates that the terminal has moved from the invoked Access Bridge.

OMG IDL 0

A.1 MobileTerminal.idl

```
//File: MobileTerminal.idl
#ifndef _MOBILE_TERMINAL_IDL_
#define _MOBILE_TERMINAL_IDL_
#include <orb.idl>
#include <IOP.idl>
#pragma prefix "omg.org"
module MobileTerminal {
    interface HomeLocationAgent;
    interface AccessBridge;
    typedef sequence<octet>  TerminalId;
    typedef sequence<octet>  GIOPEncapsulation;
    typedef sequence<octet>  GTPEncapsulation;
    struct Version {
        octet major;
        octet minor;
    };
    struct ProfileBody {
        Version                                 mior_version;
        octet                                   reserved;
        TerminalId                              terminal_id;
        sequence<octet>                         terminal_object_key;
        sequence<IOP::TaggedComponent>          components;
    };
    struct HomeLocationInfo {
        HomeLocationAgent  agent;
    };
    struct MobileObjectKey {
        Version             mior_version;
        octet               reserved;
        TerminalId          terminal_id;
        sequence<octet>     terminal_object_key;
    };
    enum HandoffStatus {
        HANDOFF_SUCCESS,
        HANDOFF_FAILURE,
        NO_MAKE_BEFORE_BREAK
};
const octet  TCP_TUNNELING = 0;
const octet  UDP_TUNNELING = 1;
const octet  WAP_TUNNELING = 2;
struct GTPInfo {
        Version         gtp_version;
        octet           protocol_level;
        octet           protocol_id;
                                //  values 0xE0...0xFF are reserved for internal use
};
struct AccessBridgeTransportAddress {
        GTPInfo                 tunneling_protocol;
        sequence<octet>         transport_address;
};
typedef sequence<AccessBridgeTransportAddress>
        AccessBridgeTransportAddressList;
typedef string ObjectId; // same as CORBA::ORB::ObjectId
typedef sequence<ObjectId> ObjectIdList;
                        // same as CORBA::ORB::ObjectIdList
exception IllegalTargetBridge {};
exception TerminalNotHere {};
exception UnknownTerminalId {};
exception UnknownTerminalLocation {};
exception InvalidName{}; // same asCORBA::ORB::InvalidNam
interface HomeLocationAgent {
        void update_location (
                in TerminalId           terminal_id,
                in AccessBridge         new_access_bridge
        ) raises (UnknownTerminalId, IllegalTargetBridge);
        boolean deregister_terminal (
                in TerminalId           terminal_id,
                in AccessBridge         old_access_bridge
        ) raises (UnknownTerminalId);
        void query_location (
                in  TerminalId                  terminal_id,
                out AccessBridge                current_access_bridge
        ) raises (UnknownTerminalId, UnknownTerminalLocation);
        ObjectIdList list_initial_services ();
        Object resolve_initial_references (
                in ObjectId  identifier
        ) raises (InvalidName);
};
interface HandoffCallback {
        void report_handoff_status (
                in HandoffStatus  status
        );
};
interface AccessBridge {
        ObjectIdList list_initial_services ();
        Object resolve_initial_references (
                 in ObjectId  identifier
        ) raises (InvalidName);
        boolean terminal_attached (
            in TerminalId  terminal_id
        );
        void get_address_info (
            out AccessBridgeTransportAddressList  transport_address_list
        );
        void start_handoff (
            in TerminalId                terminal_id,
            in AccessBridge              new_access_bridge,
            in HandoffCallback           handoff_callback_target
        );
        void transport_address_request (
            in  TerminalId                              terminal_id,
            out AccessBridgeTransportAddressList new_access_bridge_addresses,
            out boolean                                 terminal_accepted
        );
        void handoff_completed (
            in TerminalId          terminal_id,
            in HandoffStatus       status
        );
        void handoff_in_progress (
            in TerminalId          terminal_id,
            in AccessBridge        new_access_bridge
        );
        void recovery_request (
            in  TerminalId               terminal_id,
            in  AccessBridge             new_access_bridge,
            in  unsigned short           highest_gtp_seqno_received_at_terminal,
            out unsigned short           highest_gtp_seqno_received_at_access_bridge
        ) raises (UnknownTerminalId);
        void gtp_to_terminal (
            in TerminalId                terminal_id,
            in AccessBridge              old_access_bridge,
            in unsigned long             gtp_message_id,
            in GTPEncapsulation          gtp_message
        ) raises (TerminalNotHere);
        void gtp_from_terminal (
            in TerminalId                terminal_id,
            in unsigned long             gtp_message_id,
            in GTPEncapsulation          gtp_message
        ) raises (UnknownTerminalId);
        void gtp_acknowledge (
            in unsigned long             gtp_message_id,
            in unsigned long             status
        );
        void handoff_notice (
            in TerminalId          terminal_id,
            in AccessBridge        new_access_bridge
        );
        void subscribe_handoff_notice (
            in TerminalId          terminal_id,
            in AccessBridge        interested_access_bridge
        ) raises (TerminalNotHere);
    };
};
```

```
#endif
```

A.2 Module MobilityEventNotification

```
//File: MobileTerminalNotification.idl
#ifndef _MOBILE_TERMINAL_NOTIFICATION_IDL_
#define _MOBILE_TERMINAL_NOTIFICATION_IDL_
#include <orb.idl>
#include <IOP.idl>
#include "MobileTerminal.idl"
#pragma prefix "omg.org"
module MobileTerminalNotification {
    struct HandoffDepartureEvent {
        MobileTerminal::TerminalId          terminal_id;
        MobileTerminal::AccessBridge        new_access_bridge;
    };
    struct HandoffArrivalEvent {
        MobileTerminal::TerminalId          terminal_id;
        MobileTerminal::AccessBridge        old_access_bridge;
    };
    struct AccessDropoutEvent {
        MobileTerminal::TerminalId  terminal_id;
    };
    struct AccessRecoveryEvent {
        MobileTerminal::TerminalId  terminal_id;
    };
    struct TerminalHandoffEvent {
        MobileTerminal::AccessBridge  new_access_bridge;
    };
    struct TerminalDropoutEvent {
        MobileTerminal::TerminalId  terminal_id;
    };
    struct TerminalRecoveryEvent {
        MobileTerminal::TerminalId  terminal_id;
    };
};
#endif
```

A.3 Module GTP GIOP Tunneling Protocol

```
//File: GTP.idl
#ifndef _GTP_IDL_
#define _GTP_IDL_
#include "MobileTerminal.idl"
#pragma prefix "omg.org"
module GTP {
    struct GTPHeader {
        octet               gtp_msg_type;
        octet               flags;
        unsigned short      seq_no;
        unsigned short      last_seq_no_received;
        unsigned short      content_length;
    };
```

```
 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_seq_no_received;
    } last_access_bridge_info;
    unsigned long  time_to_live_request;
};
typedef RecoveryRequestBody NetworkRequestBody;
typedef RecoveryRequestBody TerminalRequestBody;
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;
};
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_seq_no_received;
    } old_access_bridge_info;
    unsigned long  time_to_live_reply;
```

```
        };
        typedef RecoveryReplyBody NetworkReplyBody;
        typedef RecoveryReplyBody TerminalReplyBody;
        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;
        };
        struct ReleaseTunnelRequestBody {
            unsigned long  time_to_live;
        };
        struct ReleaseTunnelReplyBody {
            unsigned long  time_to_live;
        };
        struct HandoffTunnelRequestBody {
            MobileTerminal::AccessBridgeTransportAddressList
    new_access_bridge_transport_address_list;
        };
        struct HandoffTunnelReplyBody {
            MobileTerminal::HandoffStatus  status;
        };
        struct OpenConnectionRequestBody {
            GIOP::TargetAddress   target_object_reference;
            unsigned long         open_connection_request_id;
            unsigned long         timeout;
        };
        enum OpenConnectionStatus {
            OPEN_SUCCESS,
            OPEN_FAILED_UNREACHABLE_TARGET,
            OPEN_FAILED_OUT_OF_RESOURCES,
            OPEN_FAILED_TIMEOUT,
            OPEN_FAILED_UNKNOWN_REASON
        };
        struct OpenConnectionReplyBody {
            unsigned long               open_connection_request_id;
            OpenConnectionStatus        status;
            unsigned long               connection_id;
        };
        struct CloseConnectionRequestBody {
            unsigned long  connection_id;
        };
        enum CloseConnectionStatus {
            CLOSE_SUCCESS,
            CLOSE_FAILED_INVALID_CONNECTION_ID,
            CLOSE_FAILED_UNKNOWN_REASON
        };
        struct CloseConnectionReplyBody {
            unsigned long               connection_id;
            CloseConnectionStatus       status;
        };
        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
        };
        struct ConnectionCloseIndicationBody {
            unsigned long                   connection_id;
            ConnectionCloseReason           reason;
        };
        struct GIOPDataBody {
            unsigned long                               connection_id;
            unsigned long                               giop_message_id;
            MobileTerminal::GIOPEncapsulation           giop_message;
        };
        enum DeliveryStatus {
            DELIVERY_FAILED_INVALID_CONNECTION_ID,
            DELIVERY_FAILED_UNKNOWN_REASON
        };
        struct GIOPDataErrorBody {
            unsigned long   giop_message_id;
            DeliveryStatus  status;
        };
        struct GTPForwardBody {
            MobileTerminal::AccessBridge                access_bridge_reference;
            unsigned long                               gtp_message_id;
            MobileTerminal::GTPEncapsulation            gtp_message;
        };
        enum ForwardStatus {
            FORWARD_SUCCESS,
            FORWARD_ERROR_ACCESS_BRIDGE_UNREACHABLE,
            FORWARD_ERROR_UNKNOWN_SENDER,
            FORWARD_UNKNOWN_FORWARD_ERROR
        };
        struct GTPForwardReplyBody {
            unsigned long  gtp_message_id;
            ForwardStatus  status;
        };
        enum ErrorCode {
            ERROR_UNKNOWN_SENDER,
            ERROR_PROTOCOL_ERROR,
            ERROR_UNKNOWN_FATAL_ERROR
        };
        struct ErrorBody {
            unsigned short          gtp_seq_no;
            ErrorCode               error_code;
        };
    };
    #endif
```

Conformance

This specification has five conformance points: One for Home Location Agent and two for both Access Bridge and Terminal Bridge. The two conformance points for bridges correspond the levels of the GIOP Tunneling Protocol. The GTP Level indicates whether (Level 2) or not (Level 1) handoff is supported.

All products compliant to this specification must support the Mobile IOR as specified in Chapter 3.

0.1 Home Location Agent

A Home Location Agent product compliant to this specification must implement all operations specified in the **HomeLocationAgent** interface (see Chapter 4):

- **update_location**
- **deregister_terminal**
- **query_location**
- **list_initial_services**
- **resolve_initial_reference**

0.2 Access Bridge

0.2.1 Level 1

An Access Bridge product compliant to this specification must implement GIOP Tunneling Protocol version 1.0 Level 1 and TCP, UDP and/or WAP Tunneling as described in Chapter 7. The product must also implement the following operations specified in the **AccessBridge** interface (see Chapter 5):

- **list_initial_services**
- **resolve_initial_reference**
- **terminal_attached**

- **get_address_info**

The product must also acts as a relay between an ORB server and an ORB client fulfilling the message processing requirements of Section 5.3, "Message Processing," on page 5-2.

B.2.2 Level 2

An Access Bridge may provide notifications of mobility related events through the **NetworkMobilityChannel** (Section 5.3, "Message Processing," on page 5-2) and support handoff.

An Access Bridge implementation supporting handoff MUST implement the GIOP Tunneling Protocol version 1.0 Level 2 (Chapter 7) as well as the handoff and access recovery procedures and the mechanisms to GTP messaging forwarding and terminal tracking as described in Chapter 8 for an Access Bridge in any of its possible role.

The **HandoffCallback** interface and the following operations specified in the **AccessBridge** interface (Chapter 8) must be implemented:

- **start_handoff**
- **transport_address_request**
- **handoff_completed**
- **handoff_in_progress**
- **recovery_request**
- **gtp_to_terminal**
- **gtp_from_terminal**
- **gtp_acknowledge**
- **handoff_notice**
- **subscribe_handoff_notice**

2.3 Terminal Bridge

2.3.1 Level 1

A Terminal Bridge product compliant to this specification must implement GIOP Tunneling Protocol version 1.0 Level 1 and TCP, UDP and/or WAP Tunneling as described in Chapter 7.

B.3.2 Level 2

A Terminal Bridge may provide notifications of mobility related events through the **TerminalMobilityChannel** (Section 6.1, "Mobility Event Notifications," on page 6-1) and support handoff.

A Terminal Bridge implementation supporting handoff MUST implement the GIOP Tunneling Protocol version 1.0 Level 2 (Chapter 7) as well as the handoff and access recovery procedures as described in Chapter 8 for the Terminal Bridge.