Supporting CORBA Applications in a Mobile Environment

Mads Haahr, Raymond Cunningham and Vinny Cahill
Distributed Systems Group
Department of Computer Science
Trinity College Dublin
Ireland
http://www.dsg.cs.tcd.ie/

Abstract

CORBA, the Common Object Request Broker Architecture, defines a framework for developing object-oriented distributed applications. Unfortunately, current implementations of CORBA have not been designed with support for mobile computers in mind. Using CORBA in a mobile environment raises a number of problems due to hardware mobility and the characteristics of wireless networks. This paper identifies and discusses these problems and presents the design and implementation of our Architecture for Location Independent CORBA Environments (ALICE). ALICE allows CORBA objects running on mobile devices to interact transparently with objects hosted by off-the-shelf CORBA implementations. Importantly, ALICE allows server as well as client objects to reside on mobile hosts without relying on a centralised location register to keep track of their whereabouts.

1 Introduction

CORBA, the Common Object Request Broker Architecture [7], from the Object Management Group (OMG), defines a framework for developing object-oriented distributed applications. CORBA is based on the client-server paradigm and the most important component in the architecture is the Object Request Broker (ORB) which is responsible for relaying object invocations from clients to server objects.

Initially, the CORBA standard made no provision for interoperability between ORBs supplied by different vendors. Later versions of the standard addressed this issue by defining a standard protocol for inter-ORB communication which is known as the General Inter-ORB Protocol (GIOP) [7, Chapter 13] and which can be mapped onto different underlying transports. The OMG also defined a mapping of GIOP onto TCP/IP known as the Internet Inter-ORB Protocol (IIOP) [7, Chapter 13]. IIOP enables invocations to be relayed between different ORBs over TCP/IP and must be supported by all CORBA 2 compliant ORBs.

ACM COPYRIGHT NOTICE. Copyright © 1999 by the Association for Computing Machinery, Inc. Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Publications Dept, ACM Inc., fax +1 (212) 869-0481, or permissions@acm.org.

Current CORBA technology, including the IIOP protocol, is not designed for use in a mobile computing environment. Using CORBA in such an environment raises a number of problems due to hardware mobility and the characteristics of wireless networks that have yet to be addressed by the OMG [6]. This paper identifies and discusses the problems of mobile CORBA and presents the design and implementation of our Architecture for Location Independent CORBA Environments (ALICE) which allows CORBA applications running on mobile devices to communicate transparently with standard CORBA applications (such as those supported by off-the-shelf ORBs) using IIOP. The architecture allows server as well as client objects to reside on mobile hosts without relying on a centralised location register to keep track of their whereabouts. IIOP clients and servers residing on mobile hosts are able to interact with IIOP servers and clients on the wired network using standard IPv4 and without requiring the wired clients and servers to know that they are interacting with clients and servers on a mobile host. In particular, no support for Mobile IP [2] is required.

The Mobile Environment

Current state-of-the-art mobile computers—laptops and personal digital assistants (PDAs) such as Windows CE devices and the Palm Pilot—are often equipped with several communication interfaces. Common types include wired and wireless LANs, digital and analogue modems, infrared links, and serial lines. Most mobile computers support two or more of these and use them at various times depending on the user's preferences as well as on his or her work and movement patterns. A common characteristic of these interfaces is that they of-

fer low bandwidth and/or low quality connections compared to traditional wired networks. In addition, the difference in cost of using the various interfaces may vary dramatically. For example, the expense of using a GSM phone is substantially higher than that of using a corporate LAN. Sometimes, a mobile device does not connect directly to a LAN but to another host that has a permanent LAN connection. A common example is a PDA connected to a desktop computer via a docking cradle. At other times, a mobile device may be directly connected to the LAN via its own network interface. A mobile host may be connected to different desktop computers and different LANs at different points in time.

In short, the networking options for a mobile host are more complex than those of a fixed host. For distributed applications designed with more static network conditions in mind (such as CORBA middleware), this environment poses a substantial challenge. The extra functionality required to deal with this environment can either take the form of mobility-enhanced applications or of special mobility support on the mobile hosts, or both.

Another problem is that the processing power and memory resources available on many mobile devices are limited in comparison to those of typical desktop machines. This restricts the user of a mobile device in that only a limited number of applications may be available. Moreover, the functionality of available applications is often limited. These limitations also affect the application developer, as the onus is on him/her to maximise the use of the available resources.

A third problem, associated with mobility rather than network connectivity or hardware limitations, is how to locate mobile devices. A mobile device may be moving from one point of attachment to another, while a host on the wired network is attempting to send data to the old point of attachment. This problem is addressed in Mobile IP but not in IPv4.

Where to Address Mobility

The problems caused by mobility can be solved on different levels in the protocol stack. Alice uses a session layer type approach in conjunction with application support. Another approach, adopted in Mobile IP, is to solve the problem at the transport layer by extending the transport protocol. There are advantages and disadvantages to both approaches.

Solving the problems at the transport layer hides mobility from higher layers. This is a general and attractive solution because all applications running on mobile devices can benefit from it. The primary disadvantage of changing the transport protocol is that it requires all the involved parties to use the modified transport protocol. Solving the problem on a higher level (such as the session layer) is a less general solution because it requires applications to use the session layer instead of the transport layer. The advantage is that no modifications to the transport protocol are required.

IIOP as Mobile CORBA

In a CORBA context, objects running on mobile hardware move along with the hardware. A CORBA object is typically hosted by an ORB but current ORBs are generally too big and cumbersome to run on the full range of current mobile devices. A better way of letting mobile applications use CORBA technology is to bring only a subset of ORB functionality onto the mobile host. The IIOP protocol is an example of such a subset. IIOP implements the minimum ORB functionality required for objects running on a mobile device to interact with remote objects.

IIOP is a client-server based protocol. The client connects to the server, sends requests and receives replies whereafter the connection is closed. A common misconception is that servers are always large and complex pieces of software which would rarely need to reside on a mobile host. In practice, however, typical distributed applications often consist of many objects, each being a client as well as a server. Therefore, it is important to support servers as well as clients on mobile devices.

Unfortunately, like most existing CORBA standards, IIOP is designed for a fairly static environment and using it in a mobile setting is not straightforward. Our work on mobile CORBA has revealed a number of problems, some of which are related to mobility and some to the characteristics of wireless networks and mobile devices.

1. It is assumed that IIOP servers rarely (or never) change their transport connection endpoints, i.e., DNS names and IP addresses.

2. Both IIOP and transport connections are assumed to break very rarely. IIOP is heavily connection-oriented but has no support for resuming a broken IIOP connection over a different transport connection. When a transport connection breaks, the IIOP connection's state is irrevocably lost. This will typically result in the states of the client and server becoming inconsistent.

3. Because IIOP assumes a single underlying transport connection for the lifetime of an IIOP connection, there is no means of changing network interface (e.g., from GSM to Ethernet) during an IIOP connection without breaking it.

4. Transport connections are assumed to have a relatively high bandwidth. As pointed out by [9], the IIOP encoding format is designed to be easy to use rather than to optimise bandwidth utilisation.

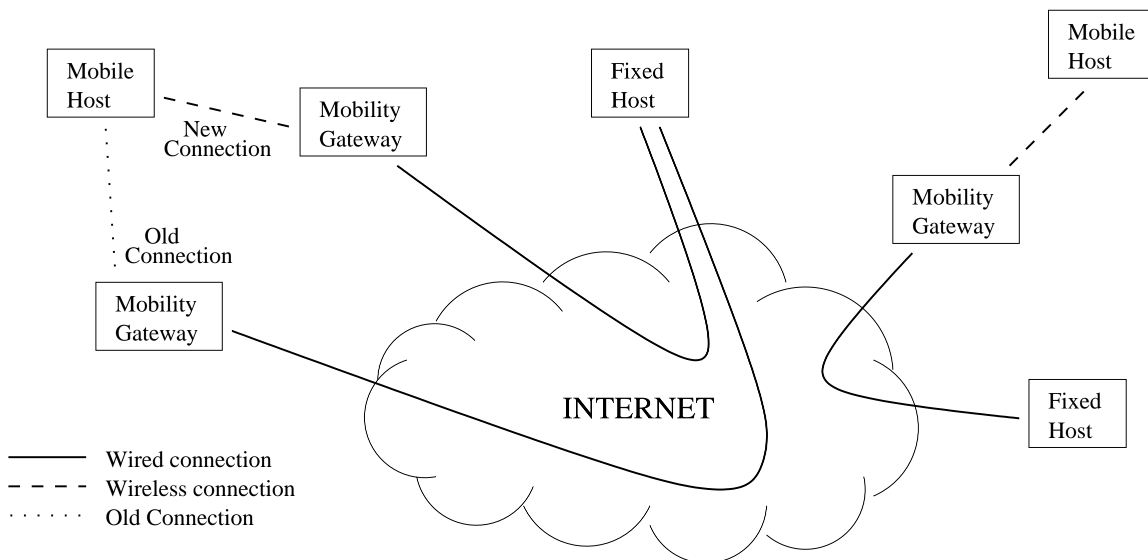

Figure 1: Communications in the ALICE Environment

The rest of this paper describes our Architecture for Location Independent CORBA Environments (ALICE) that addresses the above problems. The architecture differs from previous work such as [4] and [9] in that it supports mobile servers without relying on a centralised service to keep track of their current locations.

We proceed as follows. First, section 2 gives an overview of the architecture. Then, sections 3 to 5 describe the operation and design of the three central components of the architecture in detail. Section 6 presents the results obtained with the implementation and section 7 discusses related work. Finally, section 8 concludes.

2 Overview

This section gives an overview of the ALICE architecture. Though specific to IIOP, the ALICE architecture itself is an instance of a more general architecture which can be used for a variety of protocols. Work on this architecture is currently ongoing and ALICE, being the first in a series of implementations, has been our test case. We begin by describing the physical environment assumed by our work before presenting the software components that constitute ALICE.

2.1 The ALICE Environment

Figure 1 gives an overview of communications in the ALICE architecture. Mobile hosts are connected to *mobility gateways* via wireless links (or low-speed wired links such as serial lines) shown with dashed lines. The mobility gateway has several roles, one of which is to act as a proxy for a mobile host, relaying incoming and outgoing communications over wired connections as shown with the solid lines. Another role is to perform *address translation* and *redirection* for the higher layers, as explained in section 5.

A mobile host can change mobility gateway as it moves, causing a *handoff* from the old to the new mobility gateway, as shown to the left in the figure. This involves transferring state information from the old to the new mobility gateway and tunneling open connections for the remainder of their lifetime. Handoff, a fairly complicated procedure, is explained in detail in section 4.2.

2.2 Software Architecture

Figure 2 gives an overview of the ALICE architecture. The layers in the figure are shown in the traditional manner, such that layers at the same level communicate which each other via the layers below them. The *TCP/IP Layer* represents any implementation of the well-known protocol. Note that there is no requirement for Mobile IP to be available on the various hosts.

Apart from TCP/IP, the architecture consists of three other layers. Of these, the *Mobility Layer* (ML) provides mobility support that is independent of both CORBA and IIOP and that can also be used to support other protocols such as HTTP. The *IIOP Layer* implements the IIOP protocol independently of mobility and can be layered either above a standard implementation of TCP/IP for use in a traditional Internet environment, or above the ML for use in a mobile environment supporting client objects on mobile devices, or above the S/IIOP layer where both client and server objects are to be hosted on mobile devices. The *Swizzling* or *S/IIOP*

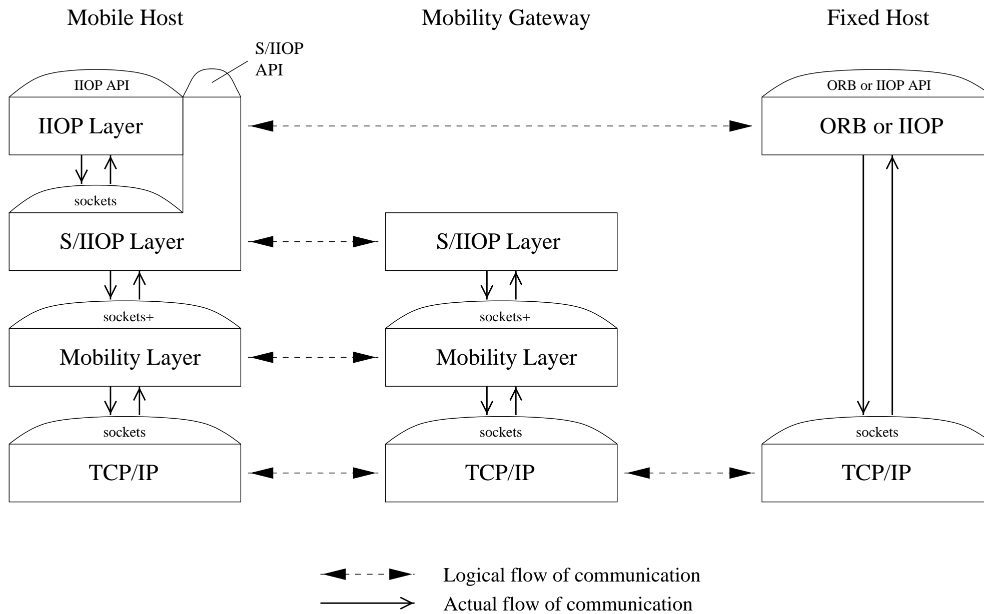

Figure 2: Overview of the ALICE Architecture

Layer provides the IIOP support that is required specifically in mobile environments where server objects are to be hosted on mobile devices.

The ML plays several roles in the architecture. First, it hides broken TCP connections from the layer above it by performing transparent reconnection attempts. In an IIOP context, this assures at-most-once invocation semantics even in the presence of broken wireless connections. Second, the ML on the mobile host lets the layer above it allocate TCP/IP ports on the mobility gateway for incoming connection attempts. This is necessary to allow clients on the wired network to create TCP connections to the mobile device. Such connection attempts are sent to the mobility gateway which creates corresponding logical connections to the mobile device. Third, it performs *handoff* between mobility gateways, in case the mobile host moves from one gateway to another. Finally, it can (optionally) notify higher layers about the current network connection point. In particular, this information is used by the S/IIOP layer to perform the object reference translation described below. The interface exported by the ML is a superset of the well-known Berkeley sockets interface providing extensions to support mobile-aware clients while still being backwards compatible with applications that use a standard sockets interface.

The S/IIOP layer is the mobility-aware component of the IIOP implementation and is used in tandem with the IIOP layer to support server objects on the mobile host. The S/IIOP layer is used by the IIOP layer to perform operations which are affected by mobility, especially publication and encoding of object references. In CORBA, each server object has its own object reference, called an Interoperable Object Reference (IOR), that uniquely identifies and locates the object. At least one $(hostname, port\#)$ pair is part of the IOR. When an IOR is created on a mobile host, the $(hostname, port\#)$ pair of the mobile host is replaced by that of the mobility gateway. Such an IOR is said to be *swizzled*. S/IIOP on the mobile host uses the underlying ML to obtain information about the current network connection point in order to perform this swizzling of IORs. This allows a client on the fixed network to contact the mobility gateway instead of the mobile host. S/IIOP on the mobility gateway is in turn configured to forward incoming requests to the server object on the mobile host. S/IIOP exports a traditional sockets-like interface to the layer above as well as operations to create and destroy object references.

The IIOP layer is our implementation of the IIOP protocol. It allows the layer above it to communicate with other CORBA applications, such as those supported by CORBA 2 compliant ORBs or other IIOP implementations, e.g., IONA's IIOP Engine [8]. The implementation expects a standard sockets interface from the layer below and can be supported directly above TCP/IP, the ML or S/IIOP layer as required.

3 The IIOP Layer

IIOP defines the minimum protocol necessary to transfer invocations between ORBs. IIOP makes a distinction between clients and servers in a request/reply interaction. A client creates an IIOP connection to a server and sends request messages to which the server typically responds with a corresponding reply message.

The client is prohibited from sending reply messages as is the server from sending request messages. The server may act as a client by opening a different connection to another server.

IIOP specifies eight message types that provide the capability to transparently locate and invoke the methods of a server object. A method is invoked using an IIOP `Request` message. The `Request` message identifies the object being invoked and also contains the symbolic name of the method. In addition, the `Request` message contains any parameters passed to the method. Any results from the invocation are returned in an IIOP `Reply` message. A client can cancel an outstanding request by sending a `CancelRequest` message to the server. In case the server object no longer resides at the location specified in the IOR, the server can return a `LOCATION_FORWARD` reply containing a new IOR for the server object.

IIOP also provides a `LocateRequest` message which can be used to check an object's location before proceeding to invoke its methods. The server replies with a `LocateReply` message containing the current location of the object in the form of a new IOR.

IIOP messages are transmitted using a well-defined transfer syntax called the Common Data Representation (CDR). CDR maps data types into a low-level representation for "on the wire" transfer between clients and servers. Simple as well as complex data types (including IORs) can be marshalled into CDR format. IORs can also be *stringified*, meaning marshalled into a string form. A stringified IOR can be transmitted via non-CORBA means such as being written to a file, sent by email or published on a web page.

3.1 Interface

Although the OMG has defined IIOP as its standard protocol for ORB interoperability, there is no standard API for IIOP implementations. A primary design consideration for our IIOP layer was to provide a consistent, object-oriented and easy-to-use API. Despite the fact that there are only eight IIOP messages, there is a fair amount of complexity (such as sequence numbering and data alignment) involved in creating and handling messages. Our API hides a lot of that complexity from the application without impairing the functionality of the protocol.

We first wrote the IIOP API in OMG's Interface Definition Language (IDL) and then mapped it to a set of C++ classes which were subsequently implemented. The API is based on the concepts of *messages* and *endpoints* described below.

Messages can be *client messages* (sent by clients to servers) or *server messages* (vice versa). Some IIOP messages can be sent both ways and therefore belong to both groups. Some IIOP messages can be used to carry data and therefore have marshalling and unmarshalling functions.

Endpoints fall into two groups: *client endpoints* (owned by clients) and *server endpoints* (owned by servers). A client uses a client endpoint to send client messages and receive server messages. Analogusly, a server uses a server endpoint to receive client messages and send server messages. An IIOP connection always has one endpoint of each type.

3.2 Design

The design of the IIOP layer can be broken into four sections: *message representation, data marshalling, transport classes,* and *communication endpoints.* Each section is discussed below.

Message Representation

A C++ class is used to represent each IIOP message. These classes inherit from either a client message or a server message class. This prevents an application, acting as a server, from sending IIOP messages which are specific to clients and a client application from sending server messages. Two of the eight IIOP messages (`MessageError` and `Fragment`) can be sent by clients as well as servers and therefore inherit from both classes.

Data Marshalling

The data-carrying IIOP messages (`Request`, `Reply`, `LocateReply` and `Fragment`) implement marshalling methods that insert the various data types into an internally managed buffer according to the data alignment requirements of IIOP. This buffer can then be sent over a transport connection. When an IIOP message is received, corresponding unmarshalling methods can be used to extract data from the IIOP message. Message classes inherit marshalling operations from a class called `CDR`.

Transport Classes

Endpoints are implemented as an abstract base class from which the subclasses `TcpEndpoint`, `MobileEndpoint` and `SwizzleEndpoint` inherit. The `TcpEndpoint` class uses the underlying TCP/IP layer while `SwizzleEndpoint` and `MobileEndpoint` use the ML. The majority of methods of the `SwizzleEndpoint` class fall through to the `MobileEndpoint` class with the exception of the `Listen()` method that implements the sockets `listen()` call, as discussed in section 5.2. Distinguishing between the underlying transport layer in this way makes it potentially possible to switch dynamically

between using the ML and the TCP/IP layer. This requires that state information is transferred from one layer to another (obviously non-trivial) and is currently being investigated.

Communication Endpoints

Client endpoints are implemented by the `ClientEndpoint` class which has three public methods: `Connect()`, `Send()` and `Receive()`. The first of these takes an instance of the `IOR` class as a parameter and uses the addressing information contained in it to create an underlying connection to the server object. The `Send()` method is used to send IIOP client messages to the server after the connection has been obtained. The `Receive()` method is used to receive an IIOP server message from the underlying connection and return a corresponding server message object to the caller.

Similar to the `ClientEndpoint` class, the `ServerEndpoint` class also has methods to send and receive IIOP mesages. Again the server is prevented from sending IIOP client messages because of the inheritance rules. In addition, the `ServerEndpoint` class provides a method to block on the currently open connections, waiting for an IIOP message to be received.

4 The Mobility Layer

From the point of view of higher layers, the ML performs four important functions.

1. It shields the IIOP layer from the inherent unreliablity of wireless media by transparently reestablishing broken transport connections either via the same, or a different, mobility gateway.

2. It lets the IIOP layer on the mobile host allocate TCP ports on the mobility gateway to accept incoming connections.

3. It offers mobility information to the S/IIOP layer on both the mobile host and the mobility gateway, so that address translation and request forwarding can be performed.

4. It performs *handoff* between mobility gateways, in particular tunnelling the open transport connections between fixed hosts and the old mobility gateway for the remainder of their lifetime.

The following sections describe the ML interface and its implementation in detail.

4.1 Interface

In order to make its functionality available to applications in an easily usable manner, the ML implements a sockets-like API known as *sockets+*. This API offers all the conventional sockets calls in addition to two new ones. The semantics one of the standard calls have been modified slightly because the ML cannot make the same guarantees with regards to interface and port allocation that an ordinary TCP implementation can.

Callbacks

The sockets+ API introduces two new operations to provide mobility information to higher layers by *registration* and *deregistration* of callback functions. When a mobile host changes mobility gateways, all registered callback functions are invoked by the ML on both the mobile host and the two mobility gateways. The API for registering and deregistering callback functions is:

```
typedef void (*CBF)
  (int fd, char *new_mg_name, int new_port);

int add_callback(int fd, CBF cbf);
int delete_callback(int fd);
```

Callbacks are only used for server sockets. A typical server application (such as our IIOP layer when used to implement a CORBA server) will invoke `socket()` followed by `bind()`, `listen()` and `accept()` when starting to wait for client connections. When using the sockets+ API, the server should also register a callback function between invoking `bind()` and `listen()`. This will cause it to receive a callback in case the mobility gateway changes. The thread that is listening will not be interrupted. The S/IIOP layer uses this functionality to maintain up-to-date information about the mobile host's current connection point.

Changed Sockets Semantics

A minor modification to the standard sockets semantics was required because the ML cannot (and should not) make the same guarantees concerning interface and port allocation as a normal implementation of TCP/IP. A server-type application using sockets typically uses the `bind()` operation to specify the interface and port number on which it wants to receive client connections. For example, a web-server would typically bind to port 80, because this is the port generally used by HTTP servers.

When an application on the mobile host performs a `bind()` using the ML, the operation is in reality performed on the mobility gateway rather than the mobile host. Consequently, it is impossible for the ML to honour a request for a specific local interface and a specific port. In addition, the endpoint actually obtained will change if the mobile host changes mobility gateway. Thus, endpoints are not only unpredictable but also short-lived.

For these reasons, the ML silently ignores any requests for particular interfaces and ports specified in the

`bind()` operation. This means that a naïve server may not be running at the interface and port number that it expects. This may seem like a drastic change at a first glance. However, because physical mobility requires the mobile host to change its IP address there is no way around this problem save for extending the IP protocol.[1] In the solution described here, a mobile-aware server (such as the IIOP implementation described in section 3) may use the callback functions to obtain the actual endpoint obtained.

4.2 Design

The ML consists of two components, one on the mobile host and one on the mobility gateway. The two components communicate via a single transport layer connection. All data exchanged by the two halves of the ML is sent on this connection in the form of ML PDUs. In the following, we describe the different PDU types and explain when they are used. When we are talking about the *mobile host* or the *mobility gateway* we really mean the ML on each respective side. PDUs have sequence numbers and are acknowledged by the other side upon receipt. Unacknowledged PDUs are cached on both sides such that they can be retransmitted if necessary.

Mobile Host as a Client

When a higher layer calls the `connect()` sockets function, to create a connection to a host on the wired network, the `connect()` call and associated information is cached in the ML. The `connect()` call returns indicating that the connection has been established.

When a client attempts to send or receive data for the first time using the `send()` and `recv()` functions over what appears to it to be a TCP connection, the ML on the mobile host sets up a logical connection to the ML on the mobility gateway, passing the server name and port number, cached by the `connect()` call, to the ML on the mobility gateway. The latter uses the server name and port number to create a TCP connection to the required host on the wired network. The ML on the mobility gateway responds with a logical connection identifier (LCID) that uniquely identifies the connection between the mobility gateway and the host on the wired network.

Data Transmission

The ML on the mobile host will assign a unique identifier to data passed to it for transmission. The ML caches the data, unique identifier and LCID before transmitting them to the mobility gateway. The ML on the mobility gateway acknowledges the sent items, caching the acknowledgement, and transmits the data on the TCP connection associated with the LCID to the fixed host.

Connection Reestablishment

The ML on the mobile host will detect when the underlying TCP connection is broken and is responsible for reestablishing the connection between the mobile host and the mobility gateway. This relieves the mobility gateway from having to know what interfaces are available on each mobile host and allows the ML on the mobile host to choose which interface it wishes to use to reestablish communication. Picking the most suitable interface in a given situation is non-trivial. Interfaces could for example be given priorities according to cost, reliability, bandwidth, connection setup time or power consumption. In practise, however, the 'best' interface would probably be defined by a combination of several such factors subject to variations according to the current state of the mobile device (e.g., connectivity and battery life) and to user preferences. In this case, picking the most suitable interface would involve querying a user profile and examining available system resources.

If the ML on the mobile host connects to the same mobility gateway, existing connections are merely resumed. We call this *reconnection.* In case the ML connects to a different mobility gateway, a *handoff* (described below) between the two mobility gateways takes place. In the former case, the ML on the mobile host sends a *Reconnect* message, including a unique identifier and the LCID, to the ML on the mobility gateway. The ML on the mobility gateway acknowledges the *Reconnect* message, and any unacknowledged data that was sent over the lost connection is retransmitted over the new TCP connection.

Connection Shutdown

Higher layers on the mobile host invoke the ML's `close()` function to close down a logical connection. Both halves of the ML retransmit any unacknowledged data until all data is acknowledged. The ML on the mobile host then sends a shutdown logical connection message to the mobility gateway. The ML on the mobility gateway removes all data associated with the logical connection and acknowledges the shutdown message. On receipt of the shutdown acknowledgement, the ML on the mobile host removes all data associated with the logical connection.

Mobile Host as a Server

When the ML `bind()` function is called on the mobile host, specifying an address and port number to which to bind, the ML caches the address and port

[1] This solution is adopted in Mobile IP. [2]

number. This is again done to minimise the use of the wireless link. When the `listen()` function is subsequently called on the mobile host, followed by a call to `accept()` or `select()`, the ML on the mobile host sends a message to the ML on the mobility gateway, to start listening for connection attempts. The ML on the mobility gateway dynamically allocates a port and acknowledges the previous message, passing back the port number and the address of the mobility gateway. The ML on the mobile host takes the dynamically allocated port and proceeds to invoke any registered callback functions associated with this logical connection.

When a client on the fixed network attempts to setup a connection with the server application on the mobile host, it must possess the *(hostname, port#)* pair of the mobility gateway. The ML on the mobility gateway relays connection attempts to the ML on the mobile host. The ML on the mobile host acknowledges the connection attempt and un-blocks the first caller of the `accept()` or `select()` functions (assuming there was one). The connection attempt between the mobility gateway and the mobile host includes the LCID already allocated along with a new LCID for the connection between the mobility gateway and the client on the fixed network.

Handoff

As described above, a mobile host reconnecting to the same mobility gateway causes the ML on the mobile host to send a *Reconnect* message. In case the mobility gateway is not the same, the ML on the mobile host initiates a *handoff* between the old and new mobility gateways by sending a *Handoff Request* message to the ML on the new mobility gateway. This request includes the address of the old mobility gateway and the identifiers of any logical connections that existed between the mobile host and the old mobility gateway.

The ML on the new mobility gateway acknowledges the handoff request and proceeds to request handoffs for each logical connection by setting up TCP connections over the fixed network to the old mobility gateway. The ML on the old mobility gateway updates the ML cache on the new mobility gateway, sending all sent but unacknowledged data (including their unique identifiers), any acknowledgements received and any data that has not yet been sent to the ML on the mobile host.

When the ML on the old mobility gateway has finished updating the cache on the new mobility gateway, it clears it owns cache and sends a *Finished Handoff* message to the ML on the new mobility gateway. It then invokes any registered callback functions (e.g., to update its S/IIOP layer), specifying the new mobility gateway address. The ML on the new mobility gateway sends a *Finished Handoff* message to the ML on the mobile host, which is then acknowledged.

At this stage, there may be a number of open transport connections between fixed hosts and the old mobility gateway. Each of these connections will be *tunnelled* between the old and new mobility gateways for the remainder of its lifetime. It is therefore possible that a chain of mobility gateways could exist if a mobile host moves frequently and a logical connection has a long lifetime. New connections are not tunnelled but refused by the old mobility gateway's ML. The old mobility gateway's swizzling layer, however, may redirect clients as described in section 5.

5 Swizzling Layer

The Swizzling Layer for IIOP (S/IIOP) is the mobile-aware part of the IIOP implementation and is necessary to support mobile servers. The S/IIOP layer is invoked by the application to perform operations that have to do with IORs. The S/IIOP layer uses the callback mechanism of the ML to keep track of the current mobility gateway and uses this information to swizzle IORs and keep existing ones up to date.

Swizzling IORs

An IOR contains a number of *profiles*, each specifying a location (for IIOP profiles, a hostname or address and a port number) at which the object can be reached. Each profile also contains an *object key* (in form of a string) identifying the object within the server at the given location. A CORBA server creating an IOR typically adds one profile to it for each endpoint at which the object can be reached. A client opening a connection to a server should try the profiles of the server's IOR one at a time until one succeeds.

Swizzling an IOR occurs when a new IOR is to be created and the mobile host is connected to a mobility gateway. In this case, each profile referring to a local interface is removed from the IOR, saved for later unswizzling and replaced by a profile for the S/IIOP layer on the current mobility gateway. The S/IIOP layer on the mobility gateway listens on a default port, which is known to the S/IIOP layer on the mobile host, thereby allowing swizzling to take place on the mobile host.

When the server starts listening on an IOR, a logical connection between the mobile host and the S/IIOP layer on the mobility gateway is set up. This allows connection attempts that arrive at the S/IIOP layer on the mobility gateway to be forwarded to the server application on the mobile host.

Reswizzling an IOR occurs when a mobile host moves from one mobility gateway to another. In this case, the S/IIOP layer on the mobile host receives a callback from the ML that the mobility gateway address has changed.

The S/IIOP layer reswizzles an IOR by replacing all profiles referring to the S/IIOP layer on the old mobility gateway with profiles referring to the S/IIOP layer on the new mobility gateway.

Unswizzling an IOR occurs if mobility support is removed from the protocol stack, for example in case the mobile host gets a direct connection to a LAN. In this case, all IORs known to the S/IIOP layer are unswizzled. For each IOR, the profiles referring to the S/IIOP layer on the mobility gateway are replaced by the local profiles that were saved during swizzling. Any profiles referring to remote interfaces are unchanged.

Invoking a Mobile Server

A client may hold a swizzled IOR identifying a server object that resides on a mobile host. Because the IOR is swizzled, it will contain one or more profiles identifying the S/IIOP layer on a mobility gateway rather than the mobile device itself. A client attempting to invoke an object with a swizzled IOR will go through the following steps.

1. Attempt to connect to the (*address*, *port#*) specified in the first profile. This is the S/IIOP layer on the mobility gateway to which the mobile host was connected when the IOR was created. If the mobile host is still connected to this mobility gateway, the connection attempt will succeed and any IIOP messages will be forwarded to the mobile server.

2. If the mobile host has moved and reconnected to a new mobility gateway (after a handoff between mobility gateways), the S/IIOP layer will redirect the client to the S/IIOP layer on the new mobility gateway. If the client sent a `Request`, the forwarding is done with an IIOP `LOCATION_FORWARD` reply. If the client sent a `LocateRequest`, a `LocateReply` is used. The S/IIOP layer on the old mobility gateway will already have been notified of the handoff by a callback from the ML on the old mobility gateway.

3. If the mobile device has not reconnected and no handoff has taken place, the S/IIOP layer on the mobility gateway cannot redirect the client. In this case, the S/IIOP layer passes the incoming connection to the underlying ML which in turn waits for the mobile host to reconnect to this or another mobility gateway. The client is blocked until this happens.[2]

The redirection message (in step 2) contains a new IOR for the object, containing a profile for the S/IIOP layer on the new mobility gateway. In case the mobile device has already moved on again, the S/IIOP layer on the new mobility gateway will redirect the client again. No attempts are made to keep old mobility gateways up-to-date during handoff, except of course for the two mobility gateways involved in the handoff.

The Validity of IORs

A client invoking a server must have an IOR for the server. In practice, this IOR can be obtained either by CORBA means (e.g., from a naming service) or non-CORBA means (e.g., being read from a file in stringified form). Since a server's IOR changes as it moves and we make no attempts to update IORs that have been published previously, clients may easily obtain IORs that are out of date. However, when invoking a server using an outdated IOR, the client will be redirected towards the current location of the server by the S/IIOP layers on the intermediate mobility gateways. Hence, the IORs maintained on these mobility gateways effectively consitute a chain of forwarding references. Work on the maintenance (e.g., shortening) of these chains is currently in progress. One approach, developed for migrating objects, is described in [5].

5.1 Interface

The S/IIOP layer on the mobile device has an API that is used by the IIOP layer above it and by the application. The S/IIOP layer on the mobility gateway does not have an API (it is the uppermost layer in the protocol stack) and it is assumed to be already in place along with the ML on the mobility gateway. The S/IIOP layer on the mobility gateway registers a callback function with the ML, so that it is notified whenever handoff takes place.

As can be seen from figure 2, the S/IIOP layer implements the sockets API. One of the reasons for the S/IIOP layer implementing a sockets-like API is to facilitate dynamic configuration of the protocol stack. In addition, the S/IIOP layer also implements the `SIOR` class. The `SIOR` class is very similar to the IOR class except that it creates swizzled IORs.

5.2 Design

An IOR is swizzled on the mobile host by pre-pending the (*hostname*, *port#*) pair onto the object key in each profile of an IOR that refers to a local interface. All profiles that are not associated with local interfaces are left untouched. Figure 3 illustrates that state of an IOR, with just one local interface, before and after swizzling.

It is important to notice that an IOR is constructed before the server starts to listen for connection attempts on the address specified within the IOR. This means

[2] Accepting connections while the mobile host is disconnected is a design decision which may change in the future. Another approach could be to refuse new connections. This may be a better approach for IIOP clients, if the IOR in question contains other profiles, some of which may be reachable.

IOR before swizzle

```
hostname   = "mobile.host.com"
port       = 1234
object key = "grid"
```

IOR after swizzle

```
hostname   = "mobility.gateway.com"
port       = 5004
object key = "mobile.host.com:1234:grid"
```

Figure 3: Swizzling of IORs

that the appropriate S/IIOP default port on the mobility gateway must be known at the time an IOR is created.

Another approach could have been to use the ML `listen()` function to dynamically allocate a port on the mobility gateway. The S/IIOP layer would then have to wait to receive a callback from the ML to obtain this dynamic port. However, this would cause the wireless link to be used unnecessarily when the application has only published the IOR and has not started to accept client connections.

Since a default port approach is used, the `listen()` function exported by the S/IIOP layer needs to be overridden to prevent the dynamic allocation of a port on the mobility gateway by the ML `listen()` function. The `listen()` function of the S/IIOP layer creates a logical connection to the S/IIOP layer on the mobility gateway passing it the mobile hostname and port number.

When an IIOP message, containing the object key of a swizzled IOR, is sent to the S/IIOP layer on the mobility gateway by a client on the fixed network, the S/IIOP layer on the mobility gateway strips off the hostname and port number from the object key. It then uses this hostname and port number to lookup the previously established logical connection and then forwards the IIOP message on this logical connection. If no corresponding logical connection can be found, the server on the mobile host is not yet ready to receive IIOP messages and the S/IIOP layer returns an error. When a handoff occurs, the S/IIOP layer on the mobility gateway is notified by a callback from the ML and it proceeds to redirect any new IIOP messages for that connection.

6 Evaluation

ALICE was initially implemented on Windows NT 4.0 using Visual C++ 5.0. It was then ported to Solaris using Sparc Works C++ and to Windows CE using the Windows CE Toolkit for Visual C++. Network communication was achieved using WinSock sockets on Windows NT and BSD sockets on Solaris. Work is still being done on the implementation and some features, for example handoff, are not completely implemented yet.

This section describes and discusses the experiments conducted with the architecture. The approach is to test the IIOP Layer with various combinations of the ALICE layers (ML and S/IIOP) enabled. A total of three combinations were tried, each featuring a fixed and a mobile host. The mobile host was a Handheld PC (H/PC) running Windows CE and equipped with a credit card GSM adaptor connected to a GSM phone. The fixed host was a desktop PC running Windows NT and equipped with a standard 33.6 kbps modem connected to an ordinary phone line. The fixed host acted as mobility gateway in all scenarios.

In each scenario, a client on one host invoked a server on the other by sending an IIOP request and receiving a reply of the same size as the request. This experiment was run for a number of different request/reply sizes (1, 128, 256, 384, 512, 640, 768, 896, 1024, 2048, 3072, 4096, and 5120 bytes) and the invocation times were measured. Each of these invocations was performed 100 times and the average invocation time was computed. In all scenarios, times were measured on the client side.

The results of the experiments are shown in figure 4 which gives invocation time as a function of request size. There are three plots, one for each configuration used:

IIOP shows results from an experiment in which the IIOP layer was run directly on top of TCP/IP, i.e., without any mobility support whatsoever. In this case, the client resided on the mobile host and the server on the fixed host.

IIOP+ML shows results from an experiment in which the IIOP layer was running on top of the ML, i.e., with mobility support but without server capabilities on the mobile host. Thus, the client resided on the mobile host.

IIOP+S/IIOP+ML shows results from an experiment where the IIOP layer was running with full mobility support, i.e., both the ML and the S/IIOP layer. In this scenario, the server was run on the mobile host and the client on the fixed host.

As can be seen from the figure, running the IIOP layer directly on top of TCP/IP generally gave better performance than in the two other cases. In particular for small invocations (less than approximately 2000 bytes) the cost of mobility support is substantial. One explanation of this could be that the MLs on both hosts spend resources multiplexing data onto a single transport connection. This involves some data copying, sequence numbering and maintaining a cache of unacknowledged PDUs, all of which make demands to

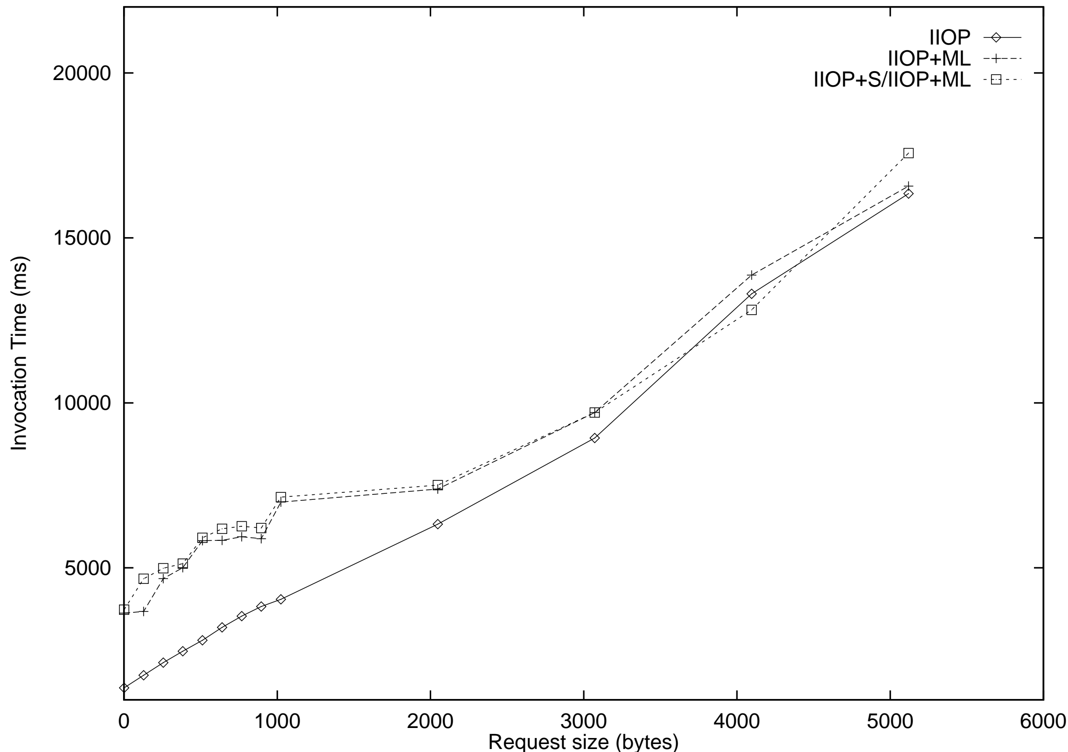

Figure 4: Performance of IIOP over a Wireless Link (Windows CE)

the mobile host's limited memory and processing power. Another factor could be the current acknowledgement scheme, (described in section 4.2) where each invocation is given its own sequence number and is acknowledged individually. For small invocations, the acknowledgement will be comparatively large as will be the time taken to transmit it. For larger invocations, the time required to transmit the acknowledgement will be relatively small compared to that of transmitting the entire invocation.

An interesting observation, which can be made from figure 4, is that the IIOP+ML and IIOP+S/IIOP+ML plots are very similar. Although using the ML introduces some overhead, it seems that little extra overhead is caused by also using the S/IIOP layer. A natural interpretation of this is that supporting mobile servers is an inexpensive addition to supporting mobile clients. It should be noted, however, that the overhead caused by the S/IIOP layer can be expected to increase as more handoffs occur and more time is spent swizzling and reswizzling IORs.

7 Related Work

This section describes related work done in the area of CORBA support on mobile devices. Two projects in particular, both supported by the European Union's ACTS programme, have addressed this problem.

DOLMEN

The DOLMEN [9] project uses a model where mobile hosts communicate with fixed hosts via *DPE bridges* much like the approach described in this paper. Mobile devices can move between bridges without losing open connections. The protocol used is called *Lightweight Inter-Orb Protocol* (LW-IOP) and is an *Environment-Specific Inter-Orb Protocol* (ESIOP). LW-IOP is functionally equivalent to IIOP but is designed with mobile environments in mind. First, it employs caching of unsent data combined with an acknowledgement scheme to deal with the unreliability of the wireless medium. Second, its object references contain not the actual names and addresses of machines (as do IORs) but references which are translated at runtime via a naming service. In this way, the DOLMEN approach supports servers on mobile hosts but relies on a register (the naming service) to maintain up-to-date information about the current location of a mobile host.

OnTheMove

Another project which solves the problem at the session layer is the OnTheMove [4] project. This project features an adaption of a commercial IIOP implementation, IONA's IIOP Engine [8], on top of a session layer. The session layer consists of two parts, one running on the mobile device and one on a base station

called a *mobility gateway*. The mobility gateway talks to the mobile host via a wireless link and the rest of the world via a wired network. The session layer shields the IIOP implementation from the unreliability of the underlying network layer but does not support handoff between mobility gateways and does not perform address translation of IORs. Therefore, servers can reside on a mobile host but when it changes connection point, clients have no means of finding its new location.

Other Projects

Other projects which deal with CORBA and mobility are the Mobiware [1] project and the Jumping Beans [3] framework. Mobiware is a toolkit which "enables adaptive mobile services to dynamically exploit the intrinsic scalable properties of mobile multimedia applications in response to time-varying mobile network conditions" [1] and is built on the CORBA distributed object model. Jumping Beans is a toolkit which lets CORBA server objects move between nodes in a network transparently to the clients. The approach is based on a centralised Jumping Beans server. When a CORBA server object moves, it first moves to the Jumping Beans server and then to its final destination.

8 Conclusion

This paper has identified and discussed the problems of mobile CORBA and presented the design and implementation of our Architecture for Location Independent CORBA Environments (ALICE). We have explained how the architecture allows both client and server objects to reside on mobile hosts and interact with standard CORBA client and server objects which are unaware that they are communicating with mobile peers. The paper has also explained why the architecture requires no mobility support from the transport layer (such as Mobile IP [2]) and how it avoids relying on a centralised location register to keep track of the mobile hosts.

The architecture has been tested in a variety of configurations over a wireless link and the results have shown that there is a certain overhead in supporting mobile clients but that mobile servers, given client functionality, come at a very low additional cost.

One particularly interesting feature of the architecture is its separation of the issues into those related to CORBA and those related to mobility. The latter are addressed in a general and CORBA-independent way by the ML, whereas the former are divided into a mobile-aware (S/IIOP) and a mobile-unaware (IIOP) part. This threefold structure is independent of CORBA and can be used as a general way of adding mobility support to some protocols (such as HTTP) in a similar fashion to the IIOP implementation described here.

Acknowledgements

The authors are very grateful to IONA Technologies plc for their generous support and to Tim Walsh and 'Tricia Garvey for lending us their mobile phones.

References

[1] Oguz Angin, Andrew T. Campbell, Michael E. Kounavis, and Raymond R.-F. Liao. The Mobiware Toolkit: Programmable Support for Adaptive Mobile Networking. *IEEE Personal Communications Magazine*, August 1998.

[2] C. Perkins (editor). RFC 2002: IP Mobility Support. Technical report, IBM, October 1996.

[3] Ad Astra Engineering. Jumping Beans White Paper. http://www.jumpingbeans.com/, December 1998.

[4] Peter Kemp et. al. *Design of MASE V2*. http://www.sics.se/~onthemove/docs/OTM_d33.doc, 1996.

[5] Robert Fowler. *Decentralized Object Finding Using Forwarding Addresses.* PhD thesis, University of Washington, Seattle, Washington 98195, December 1985. Technical Report 85-12-1.

[6] Object Management Group. Supporting Wireless Access and Mobility in CORBA, Request For Proposal. (OMG telecom/98-06-04), June 1998.

[7] Object Management Group. *The Common Object Request Broker: Architecture and Specification, V2.2.* Object Management Group, February 1998.

[8] IONA. *IIOP Engine Programming Guide.* IONA Technologies plc, 1997.

[9] P. Reynolds and R. Brangeon. Service Machine Development for an Open Long-term Mobile and Fixed Network Environment. Project deliverable, DOLMEN Consortium, December 1996.