IMPLEMENTING THE CORBA GIOP IN A HIGH-PERFORMANCE OBJECT REQUEST BROKER ENVIRONMENT

Geoff Coulson and Shakuntala Baichoo

Distributed Multimedia Research Group,
Computing Department,
Lancaster University,
Lancaster LA1 4YR,
UK

email: geoff@comp.lancs.ac.uk

ABSTRACT *The success of the Object Management Group's General Inter-ORB Protocol (GIOP) is leading to the desire to deploy GIOP in an ever-wider range of application areas, many of which are significantly more demanding than traditional areas in terms of performance. The well-known performance limitations of present day GIOP-based object request brokers (ORBs) are therefore increasingly being seen as a problem. To help address this problem, this paper discusses a GIOP implementation which has high performance and quality of service support as explicit goals. The implementation, which is embedded in a research ORB called GOPI, is modular and extensible in nature and includes novel optimization techniques which should be separately portable to other ORB environments. This paper focuses on the message protocol aspects of GOPI's GIOP implementation; higher layer issues such as marshalling and operation demultiplexing are not covered in detail. Figures are provided which position GOPI's GIOP performance against comparable ORBs. The paper also discusses some of the design decisions that have been made in the development of the GIOP protocol in the light of our implementation experience.*

Key words: Middleware, Distributed Systems, OMG CORBA, GIOP, IIOP, Protocol Engineering and Performance Optimization.

1. Introduction

In recent years, distributed middleware platforms such as the Open Group's DCE [OG,99], the Java RMI [Sun,99], Microsoft's DCOM [Microsoft,99] and the Object Management Group's CORBA [OMG,99a] have achieved both technical maturity and commercial acceptance. Among these platforms, which are also commonly known as *object request brokers* or ORBs, CORBA has to date been the clear market leader.

Much of the initial success of CORBA was due to the early standardization of its object invocation protocol, the *General Inter-ORB Protocol* (GIOP), and the latter's subsequently widespread deployment in the TCP/IP environment (GIOP over TCP/IP is referred to as the *Internet Inter-ORB Protocol* or IIOP). Subsequently, while GIOP was originally envisaged as a protocol for communication *between* CORBA ORBs from different vendors, each of which would internally use its own proprietary protocol, it has in the event been almost universally deployed *within* CORBA implementations as well as between them. Today, GIOP is a widely deployed protocol and is increasingly being used in areas beyond CORBA; for example, in the WWW environment [Sun,99].

Although GIOP is a well-designed and widely implemented protocol, it is not always implemented *efficiently* [OMG,99b,c]. This is particularly true in the case of commercially successful GIOP-based ORBs such as Inprise's Visibroker or Iona's Orbix (a version of Orbix is evaluated in section 6.4) which, according to the literature (see, for example, [CC,99]), are significantly slower than research ORBs such as TAO [Schmidt,97], OmniORB [Lo,98], or GOPI [Coulson,99a]. One reason for this performance deficit is that the commercial vendors, quite understandably, have traditionally focused on *facilities* and *reliability* rather than on performance. Indeed, optimal performance was scarcely an issue in traditional application domains such as banking and database integration. Today, however, there is an increasing desire to apply GIOP-based ORB technology in more demanding areas such as interactive, multimedia and mobile systems [Blair,97], and to deploy ever larger and more complex distributed applications. In such environments high performance becomes crucial.

This paper shares experience in efficiently implementing GIOP. More specifically, the paper

Coulson, G., Baichoo, S., "Implementing the CORBA GIOP in a High-Performance Object Request Broker Environment", *ACM Distributed Computing Journal*, Vol. 14, No. 2, pp 113-126, Springer Verlag Press, April 2001.

discusses GIOP implementation issues in the context of a high performance research ORB called GOPI [Coulson,99a]. The implementation is modular and extensible, supports flexible quality of service (QoS) configurability, and includes a number of optimization techniques that should be separately portable to other environments. Some of these techniques have been implemented elsewhere and we include them for the sake of completeness; others, to the best of our knowledge, are novel. The focus of the paper is on the low-level *messaging protocol* aspects of GIOP; we intend to address higher-level aspects such as marshalling in a forthcoming paper.

The remainder of the paper is structured as follows. Sections 2 and 3 contain essential background on the GIOP messaging protocol and on the GOPI ORB respectively. Sections 4 and 5 then discuss aspects of GOPI's GIOP implementation in detail, focusing particularly on performance engineering issues. Section 4 addresses standard IIOP implementation, while section 5 discusses non-standard extensions that enable the use of GIOP in a QoS configurable environment. Following this, section 6 presents performance measures and section 7 discusses related work. Finally, we discuss our impressions of the GIOP protocol in section 8 in the light of our implementation experience, and offer concluding remarks in section 9.

2. Background on CORBA GIOP

2.1 Architecture

The OMG's GIOP specification[1] can be viewed as comprising three distinct standards. Firstly, a *messaging standard* defines packet headers, protocols for remote communication, and requirements on the underlying transport service. Secondly, the *Common Data Representation* (CDR) standard defines on-the-wire encodings for primitive and structured data-types in messages. Finally, the specification defines the structure and content of *Interoperable Object References* (IORs) which act as location transparent object identifiers. In this section, we focus exclusively on the messaging standard. The CDR and IOR parts of the specification are not discussed further because issues relating to their implementation are not addressed in the paper.

2.2 Message Types

The GIOP messaging standard defines an object request protocol that incorporates eight message types: *Request*, *Reply*, *LocateRequest*, *LocateReply*, *CancelRequest*, *CloseConnection*, *MessageError* and *Fragment*. The definitions of these messages and the protocols governing their exchange are independent of the underlying message transport layer; it is, however, required that the underlying transport layer be *reliable*[2].

GIOP is a client-server protocol. *Request* messages, which carry all the information necessary to invoke a remote object, are sent by clients, and *Reply* messages, which are sent in response to *Request* messages, are sent by servers. Client and server roles (respectively) are similarly assigned to the *LocateRequest* and *LocateReply* messages; this pair is used to query the current location of an object. It is permissible to multiplex requests on connections; i.e. one can issue new *Request* (or *LocateRequest*) messages on a given connection before replies to previously issued requests on the same connection have been received.

The remaining four messages are self-standing rather than paired. *CancelRequest*, a client-side message, is used to advise servers that a reply is no longer required for the (still pending) request whose identifier is specified in the message. *CloseConnection* is a server-side message used to advise the client not to send further requests on the connection on which the *CloseConnection* message was received, as this connection is about to be closed. Finally, *MessageError* and *Fragment* messages can be sent by either clients or servers. The former is sent in response to any message with a bad header, and the latter is used to support multi-fragment messages. *Fragment* messages follow an incomplete preceding message (of type *Request*, *Reply*, *LocateRequest*, *LocateReply* or *Fragment*) which has its 'following fragment' bit set (see section 2.3 below). The last *Fragment* message in a multi-fragment message has its 'following fragment' bit unset.

2.3 Message Headers

All GIOP message types employ a *fixed-sized message header* (see figure 1). The *magic* field in this header is used to identify messages as GIOP messages, the *version* field specifies the GIOP

[1] Although this paper is based on version 1.1 of the specification (as this is the version currently implemented by GOPI) a more recent version, v1.2, is now available. However, the main changes in GIOP v1.2 (e.g. the option for servers as well as clients to initiate requests) primarily affect the stub/ skeleton layer; there is little impact on the essentials of the message protocol area that is the focus of this paper. Those aspects of v1.1 that do impact this paper are discussed in footnotes in section 8.

[2] Nevertheless, efforts have recently been initiated within the OMG to define GIOP-like services over unreliable protocols such as multicast IP. See http://cgi.omg.org/cgi-bin/doc?orbos/99-11-14.

protocol version and the *message type* field identifies the message's type (i.e. as one of the eight possible types described above). The *flags* field includes a bit to specify whether the sender is running on a little or a big-endian architecture and also a 'following fragment' bit to specify whether or not this message is complete or only a fragment (see section 2.2). The *message size* field contains the length of the whole message in octets, excluding the 12 octets of the fixed-sized message header itself.

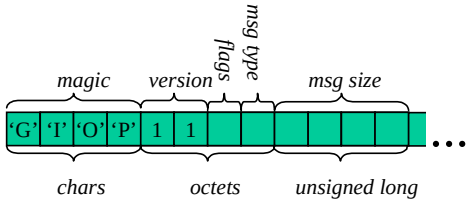

Fig. 1: The GIOP fixed-sized message header

In addition to the fixed-sized message header, all message types except *CloseConnection*, *MessageError* and *Fragment* additionally employ a *message specific* header situated between the message header and the payload. For example, the *Request* message's message specific header comprises the following sequence of fields:

- an unsigned long *l* followed by a list of *l service contexts* (*l* is 0 for an empty service context list); service contexts contain auxiliary information (e.g. a transaction or security identifier or priority information) that may need to be passed to an operation invocation; they are each encoded as an unsigned long *m* followed by *m* octets of data;
- an unsigned long containing a unique *request identifier*; this is used to match *Request*s with their corresponding replies and to identify *Request*s in *CancelRequest* messages;
- an octet interpreted as a boolean that specifies whether or not a response to this *Request* is expected;
- three octets that are currently unused but serve to pad the previous field so that the following field is appropriately aligned;
- an unsigned long *l* followed by *l* octets representing the *object key;* this is the unique identifier of the target object;
- an unsigned long *l* followed by *l*–1 octets representing the *operation string*, followed by a null octet; the operation string identifies the target operation name (this is assumed to refer to an operation supported by the target object);
- an unsigned long *l* followed by *l* octets representing the *requesting principal* (*l* is 0 for empty requesting principals); this field identifies the requesting object (e.g. for security or accounting purposes).

The fields comprising the message-specific headers of the other message types are largely subsets of the above set of fields. More specifically, the fields of the remaining message types are as follows. *Reply*: service context, request identifier and *reply status* (the latter is an unsigned long); *LocateRequest*: request identifier and object key; *LocateReply*: request identifier and reply status; *CancelRequest*: request identifier (of the request to be cancelled). In the *LocateReply* case, if the reply status field indicates success, the message payload is assumed to contain a marshalled IOR that specifies the current location of the queried object.

Note finally that the *CancelRequest* and *LocateReply* messages employ *fixed length* message-specific headers whereas the *Request*, *Reply* and *LocateRequest* headers are of *variable length* because they include variable length fields (i.e. one or more of: service context, object key, operation string or requesting principal fields).

3. Background on the GOPI ORB

3.1 Overall Architecture

The GOPI ORB architecture comprises two levels:

- the *GOPI-core* level and
- a higher-level *personality level*.

GOPI-core offers a generic support infrastructure and application programmer's interface (API) for core ORB functions. It includes a set of concurrency services and a generic communication protocol framework, both of which are QoS configurable and both of which attempt to honor QoS specifications through (configurable) resource management strategies. Personality layers build on GOPI-core to offer some particular higher-level platform-specific API. To date, a standard CORBA personality and a multimedia-capable personality, based on CORBA/ RM-ODP [ITU-T,95], have been developed [Coulson,99b], [Coulson,00].

3.2 GOPI-core

3.2.1 Architecture

GOPI-core consists of approximately 12,000 lines of C and runs on a variety of platforms including SunOS, Linux and Win32. It is implemented in an object-oriented style that could be straightforwardly translated into C++ or Java.

The software is structured as the following set of independent modules:

- the *thread* module is a sophisticated concurrency package which supports *application scheduler contexts* (ASCs). ASCs are pluggable modules which provide user-level threads with varying semantics and QoS. Each ASC is defined as a tuple: <*set of user-level threads*, *set of kernel threads*, *scheduling policy*, *QoS-schema*>, and its function is to multiplex its set of user-level threads over its kernel threads (referred to as 'virtual processors') according to its associated scheduling policy. The QoS-schema defines parameters through which the scheduling policy may be configured on a per-thread basis (e.g. the QoS-schema associated with an earliest deadline first policy may include deadline and period parameters). ASCs are highly dynamic: they can be created or destroyed at run-time and both threads and virtual processors can be added/ removed at will or migrated across ASCs [Coulson,99a].
- the *buf*/ *chan* modules respectively manage buffers and provide an efficient inter-thread buffer passing service;
- the *tp*/ *asp* modules deal with communications; *tp* provides a common abstraction layer over a selection of OS supported transport protocols and also notifies interested parties of message arrivals; *asp* then provides a framework above the transport layer for accommodating stacks of *application specific protocols* (known as ASPs; see below); the GIOP message protocol is implemented as an ASP;
- the *iref* module supports location transparent communication endpoints known as *interface references* (or *irefs*); these are employed in the implementation of IORs in the CORBA personality;
- the *iiop* module provides a set of support services for *IIOP bindings* (see section 3.2.3 below) which are layered over the GIOP ASP;
- the *bind* module supports a generic two-phase binding protocol through which *QoS bindings* between irefs can be established (again, see section 3.2.3 below).

Although only a minimally necessary amount of detail is given on GOPI-core internals in this paper, certain aspects do require further elaboration[3]. In particular, ASPs, as supported by the *asp* module, and the three styles of binding supported by the *iiop*/ *bind*/ *asp* modules, are further described in the following subsections.

3.2.2 Application Specific Protocols

ASPs are communication protocol modules (classes) that support three interfaces:

- a *data transfer* interface with calls such as *send()*, *recv()* and *call()*,
- a *connection management* interface, inspired by the *listen()*/ *connect()*/ *accept()*-style Berkeley sockets API, and,
- an (optional) *auxiliary* interface with arbitrary ASP-specific calls (e.g. the various *giop_*()* calls discussed in section 4.2.3).

Instances of each ASP are QoS configurable in terms of specifications written in an ASP-specific QoS-schema, analogous to the ASC framework's QoS-schema. Each ASP maps QoS-schema specifications passed to its connection management operations either to primitive GOPI-core resources such as threads, ASCs, buffers or transport connections, or to the QoS-schema of a further ASP or ASPs which it may choose to instantiate below itself (stacks of ASPs are created in this manner). More details on QoS mapping and negotiation are given in section 5 below.

The ASPs implemented in the current release of GOPI comprise the GIOP ASP described in this paper, an alternative request/ reply protocol called FRAG which is simpler and faster than GIOP, 'adaptive' ASPs for audio and video communications and a reliable message-oriented multicast ASP [Coulson,99a].

3.2.3 Binding Styles

GOPI supports three distinct binding styles which are classified according the nature of the 'end-points' involved in the binding; these can be either *capsules* (also known as processes or address spaces) or *irefs*.

[3] Detailed descriptions are available in [Coulson,98], [Coulson,99a], [Coulson,99b] and [Coulson,00] and definitive documentation of the GOPI-core API can be found at: http://www.comp.lancs.ac.uk/computing/users/geoff/GOPI.

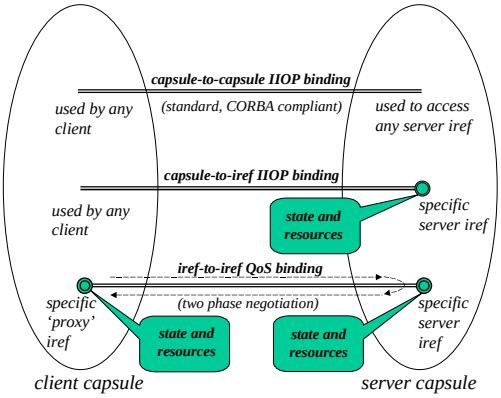

Fig. 2: Binding styles

The three binding styles, which are illustrated in figure 2, are described as follows:

- *capsule-to-capsule IIOP bindings* These are 'conventional' CORBA bindings and can be used to bind to services supported by any standard CORBA ORB. These bindings are styled 'capsule-to-capsule' because they support communication between any object in the client capsule and any iref (or CORBA IOR) in the server capsule.
- *capsule-to-iref IIOP bindings* These are 'enhanced' IIOP bindings which aim to improve performance and predictability by maintaining binding-related state and resources at the server; they are non-standard and operate only in the GOPI environment. They are styled 'capsule-to-iref' because they support communication between any object in the client capsule and a single *specific* iref (or CORBA IOR) in the server capsule.
- *iref-to-iref QoS bindings* These bindings support communication between a single specific 'proxy' iref in the client capsule and a single specific iref in the server capsule. Like capsule-to-iref IIOP bindings, they are non-standard and only operate in the GOPI environment. Unlike capsule-to-iref bindings, iref-to-iref QoS bindings hold state and resources at both ends which is *negotiated* between proxy and server irefs using the *bind* module's binding protocol. They are especially suited to stream based communication using media-specific ASPs [Coulson,99a]. However, they can also be used for request/ reply based communication, including the use of the GIOP ASP in non-IIOP configurations as discussed in section 5.

The implementation of standard (capsule-to-capsule) IIOP bindings is discussed in detail in section 4. The other two, non standard, binding styles are covered in section 5.

4. Standard IIOP Bindings

4.1 Architecture

4.1.1 CORBA Personality Level

Stubs and *skeletons* comprise the major functionality at the personality level. These are responsible for marshalling and unmarshalling C++ application data-types to and from GOPI-core buffers; stubs operate at the client side and skeletons at the server side. Both stubs and skeletons interface to the GOPI-core *iiop* module and the GIOP ASP auxiliary interface using the APIs discussed in section 4.2. Traditionally, skeletons are also responsible for *operation-level demultiplexing* - i.e. selecting the operation specified in incoming *Request* messages. However, optional support for operation-level demultiplexing is also provided at the GOPI-core level (see section 4.2.8 below).

As mentioned, the present paper focuses on the message protocol aspects of GIOP in GOPI-core and only refers in passing to personality-level functionality.

4.1.2 GOPI-core Level

The IIOP binding architecture at the GOPI-core level (i.e. excluding stubs and skeletons) is illustrated in figure 3. The heart of the architecture, the GIOP message protocol, is situated in the GIOP ASP in the *asp* module. This ASP is used in both IIOP bindings and in non-standard QoS bindings (see section 5 for discussion of its role in the latter).

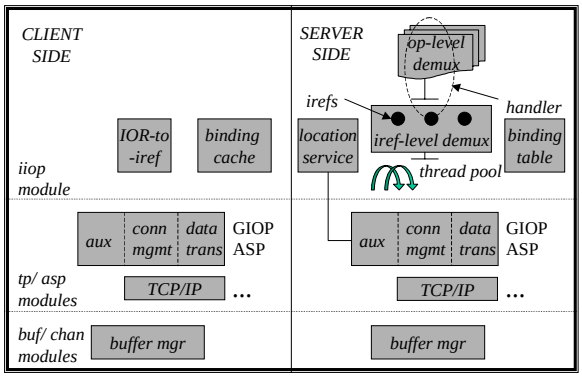

Fig. 3: The GOPI-core level IIOP binding architecture

IIOP bindings further rely on buffer management services in the *buf* module and on a range of *iiop* module services. In the client-side *iiop* module, a *binding cache* is provided for client-side IIOP bindings, and a mapping of CORBA IORs to GOPI irefs is maintained. At the server side, the following services are implemented:

- a *binding table* of current server-side IIOP bindings,

- a *thread pool* which contains threads on which incoming requests are serviced,
- an *iref-level demultiplexor* which demultiplexes incoming requests to the target iref,
- an *operation-level demultiplexor* which demultiplexes requests to the target operation (as mentioned, use of this is optional; operation level demultiplexing may also be carried out by personality-level skeletons), and
- a *location service* which resolves IOR location requests (*LocateRequest* messages) for the GIOP ASP.

With the exception of thread pool and the operation-level demultiplexor, none of the above services are discussed in detail in this paper, despite their importance in efficient GIOP implementation. For a good discussion of demultiplexing and binding cache management refer to [Gokhale,97] and [Gokhale,98]. GOPI's approach to the optimization of buffer management, together with other efficiency related concerns such as thread context switching, inter-thread communication, notification of incoming messages, use of scatter/ gather IO etc., has been discussed in [Coulson, 98] and [Coulson,99a].

4.2 Design Aspects

4.2.1 Scope of Discussion

We now discuss detailed aspects of the design of the GOPI-core IIOP binding infrastructure. The discussion focuses on noteworthy and novel aspects of the design, particularly those that significantly impact performance. The areas addressed are

- the caching of headers,
- the use of non-multiplexed connections,
- the use of a subset of GIOP while remaining GIOP conformant,
- an optimization to reduce the overhead of the OS level *recv()* call in receiving messages,
- the data path used in request handling, and
- the above mentioned scheme for efficient operation level demultiplexing.

To form a basis for the subsequent discussion, we begin the section by describing the general GOPI-core API for connection management and communications. In addition to this API, which is provided by the *iiop* module and used by stubs and skeletons, the GIOP ASP provides additional API services, to be discussed below in section 4.2.3, through its auxiliary interface.

4.2.2 Connection and Communication API

```
/* client side APIs */
int iiop_getbinding(Iref *serv,
    boolean exclusive);
int iiop_invoke(int asap,
    Buffer *req, **rep);
int iiop_putbinding(int asap);
int iiop_close(int asap);

/* server side APIs */
typedef struct {
    CharSeq serv_ctx_list[MAX_SC],
    ULONG serv_ctx_list_length,
    OCTET *object_key;
    ULONG object_key_length;
    char *operation_string;
    OCTET *req_principal;
    ULONG req_principal_length;
} ReqState;

typedef int (*Handler)(Iref *iref,
    ReqState *state,
    Buffer *req, Buffer **rep);

Iref *iref_create(Aspname asp,
    Handler h, ...);
int iiop_send(int asap, Buffer *rep);
```

Client side stubs use *iiop_getbinding()* to obtain an *ASP service access point identifier* (asap) which represents a binding to an address specified in the call's *iref* argument (the *exclusive* argument is discussed in section 4.2.4 below). When a binding is no longer required it is either returned to the binding cache using *iiop_putbinding()* or torn down using *iiop_close()*.

Before issuing an invocation on a binding, stubs use the GIOP ASP's auxiliary interface (to be discussed in section 4.2.3) to set up per-asap invocation context such as the target iref and operation name. Subsequently, the invocation is issued using *iiop_invoke()*. The *req* parameter to *iiop_invoke()* contains the request arguments, and the reply in received into **rep*.

Note that separation of the invocation process into three distinct stages (i.e. obtaining a binding, setting up the invocation context, and issuing the invocation) lends itself well to performance optimizations. In particular, the separation of binding and invocation allows stubs to hold bindings over multiple invocations, and the separation of setting the invocation context and issuing the invocation allows this context to be reused for subsequent invocations that use the same context (see section 4.2.3).

The server side personality level creates service instances (irefs) using the *iref_create()*[4] call. This

[4] Some arguments of this call are omitted here for clarity; see http://www.comp.lancs.ac.uk/-computing/users/geoff/GOPI/index.html for full details.

call specifies a *Handler* to be upcalled when requests on the new iref are received. Handlers are used to pass the request arguments up to the skeleton layer and to receive the address of a reply buffer on completion of the invocation. In addition, the associated invocation context is passed to the skeleton in a *ReqState* struct. This holds only pointers to request header data; no copy overhead is incurred. The server side replies to the invocation using *iiop_send()*.

4.2.3 Use of Cached Headers

As mentioned, before an invocation or reply is issued on a binding, services in the GIOP ASP's auxiliary interface are used to set the context of the forthcoming request or reply; i.e. to pre-select a message type and to fill in the header of the selected message type. For example, the *giop_hdrrequest()* call below pre-selects and initializes a *Request* message header. Similar calls are provided for other message types. When an invocation (or reply etc.) is issued, the pre-selected header is transmitted, along with the given request buffer, in a single scatter/ gather IO call.

```
typedef struct {
    OCTET *data;
    ULONG length;
} CharSeq;

int giop_hdrrequest(int asap,
    CharSeq serv_ctx_list[],
    ULONG serv_ctx_list_length,
    boolean response_expected,
    char *operation,
    OCTET *object_key,
    ULONG object_key_length,
    OCTET *req_principal,
    ULONG req_principal_length);
```

The following pair of services offer an alternative to *giop_hdrrequest()* which implements a useful optimization that can be employed by stubs when series of invocations are to be made on the same target object:

```
int giop_setdefaulttarget(int asap,
    OCTET *object_key,
    ULONG object_key_length
int giop_hdrdefaultreq(int asap,
    char *operation);
```

The first call of *giop_setdefaulttarget()* on a given binding allocates and caches a default *Request* header in which all fields except the operation name are pre-filled. In particular, the fixed-sized message header's *magic*, *version*, and *endian flag* fields are statically filled according to fixed implementation/ runtime environment properties, the *following fragment* flag is set to FALSE in line with our policy of not sending fragmented messages (see section 4.2.5), and the *message type* field is set to *Request*. In addition, the header's *Request*-message-specific *response expected* field is set to TRUE, the *service context* and *requesting principal* fields to empty and the *object key* field to the value provided.

Having built and cached this header, a series of invocations can subsequently be made on the same target object without incurring any additional header-related overhead[5].

The *giop_hdrdefaultreq()* call supplements *giop_setdefaulttarget()* by allowing the calling stub to inexpensively tailor the cached header to invoke a different operation on the same object. Even if *giop_setdefaulttarget()* is called again to select a new target object, all the other non-target-related pre-filled fields remain unchanged.

Similar services to the above are provided for *Reply* headers. In addition the *giop_hdrrequest()* call, and its peers, cache headers in a similar way so that some at least of the header filling overhead can be avoided (in particular, most of the fixed-sized message header's fields do not need to change from invocation to invocation).

4.2.4 Control of Multiplexing

Although the GIOP specification allows new requests to be issued on bindings on which previously issued requests are still pending (see section 2.2), our implementation does *not* encourage exploitation of this possibility. One problem with such multiplexing is that it incurs additional per-request overhead to ensure that replies are correctly demultiplexed; i.e., passed to the thread that issued the corresponding request. Other problems/ overheads are discussed in sections 4.2.6 and 8. To avoid these overheads, we prefer that concurrent invocations to the same target capsule be carried on *separate* bindings. This leaves the demultiplexing to the operating system (which, of course, needs to demultiplex anyway) and thus simplifies and streamlines the ORB implementation.

The *exclusive* argument to *iiop_getbinding()* is used to control whether or not a multiplexed connection is to be returned. If *exclusive* is true, *iiop_getbinding()* atomically removes a binding from the binding cache if it finds one there or creates a new binding if it fails to find one.

4.2.5 Use of a Subset of GIOP

Our client-side API only supports the sending of *Request* and *LocateRequest* messages, despite that fact that the GIOP standard additionally allows

[5] Of course, the cached header cannot be used for invocations for which these defaults are inappropriate. For example, the stub for a one-way *Request* would have to use the less efficient *giop_hdrrequest()* call rather than *giop_hdrdefaultreq()* to build a header with the *response expected* field set to FALSE.

Fragment and *CancelRequest* messages to be issued by clients. We do not support the sending of *Fragment* messages (i.e. we don't provide a *giop_hdr*()* service for them) because we have as yet discovered no real motivation to do so (see section 8). Furthermore, our preference for non-multiplexed bindings renders the *CancelRequest* message largely redundant as there is only ever one outstanding request per binding on a non-multiplexed binding. If it is required to cancel an outstanding request, this can simply be achieved on both non-multiplexed and multiplexed bindings by calling *iiop_close()* from a thread other than the one blocked on *iiop_invoke()*. This causes an error return from *iiop_invoke()* and a corresponding exception at the server handler, and the request (or possibly multiple requests in the case of a multiplexed binding) is thus effectively aborted.

The GIOP standard also allows that the client-side may legally receive *CloseConnection* or *MessageError* messages in addition to the expected replies to *Request* or *LocateRequest* messages. Our client-side implementation deals with these (presumably infrequent) eventualities simply by returning an appropriate error code to *iiop_invoke()*, which is ultimately passed up to the calling stub.

Our handling of *CloseConnection* messages is unorthodox in that we make no attempt to act on these messages in a *timely* manner (i.e. despite that fact that they may be issued asynchronously by the server, we receive them only incidentally when expecting a reply to a pending request). Our strategy is to allow the server to asynchronously terminate the binding and to detect and recover from this the next time the client side attempts to use the binding. This satisfies the required GIOP semantic while significantly streamlining the client-side implementation (in particular, we do not need a dedicated thread to deal with *CloseConnection* messages).

4.2.6 Message Receipt Optimization

GIOP implementations typically employ *two recv()* calls to receive each message at the OS-level: one to read the 12 octet fixed-sized message header and a second to read the remainder of the message (the length of which can be found in the fixed-size header). However, calling *recv()* twice per message incurs a non-trivial overhead[6].

Our approach to alleviating this overhead is to attempt to receive messages using only a single *recv()* call, by receiving a heuristically pre-determined number of bytes, *b*, into a pre-allocated buffer of size *b*, using a single non-blocking *recv()* call. The effectiveness of this strategy is highly dependent on the actual size, *msize*, of incoming messages. There are two cases to consider:

- with $msize > b$ we must allocate a new buffer of size *msize*, copy the already received *b* bytes into the new buffer, and issue a second *recv()* call to receive the remaining $msize - b$ bytes; i.e. we *still* incur two *recv()* calls *plus* some additional overhead due to buffer management complexities;
- however, with $msize \leq b$, we have read the whole message in one non-blocking call, thus saving the overhead of a second *recv()* call.

Clearly, the choice of *b* is critical; it should be large enough to maximize the probability of the $msize \leq b$ case, but not so large that buffer space wastage becomes an issue. In our current experimentation we are varying *b* adaptively so that its value is informed by the actual size of previously received messages.

An additional complexity is that the $msize \leq b$ case may incur a slight additional overhead in circumstances when part of a *following* message is received along with the current message, and the following message (or part thereof) must therefore be copied to a new buffer. In multiplexed bindings, this additional overhead would be incurred relatively frequently as the following message could be of any type. However, it should be incurred far less often on non-multiplexed bindings where the only message types that can follow (i.e. *Fragment*, *CancelRequest* and *CloseConnection*) are those which tend to occur relatively infrequently.

For multiplexed bindings, a potentially useful variation of the scheme is as follows[7]: The very first *recv()* call on a binding reads just enough data to obtain a fixed-length GIOP message header (12 octets). Then, all subsequent *recv()* calls read the length of data indicated in the last GIOP message header plus the size of the fixed-length GIOP message header. In this way, if there are a number of clients writing to a connection, the number of *recv()* calls per message will ideally converge to 1.

4.2.7 Request Handling

When an incoming message arrives at a server

[6] We implemented a TCP/IP based client/ server pair in which the client repeatedly sent 1024 byte packets to the server which subsequently echoed them back. A 17% overhead was observed when the programs were modified to each receive 12 bytes and then 1024–12 bytes in separate *recv()* calls as opposed to receiving the whole 1024 bytes in a single *recv()*.

[7] This variation was suggested by one of the anonymous reviewers of this paper.

capsule, its availability is detected by a *poll()* system issued by a 'sentinel' thread in the *tp* module and notified to one of a dedicated pool of user-level server threads [Coulson,99a]. The size of this pool, which is implemented as an ASC (see section 3.2.1), can be dynamically resized as a function of the number of open bindings. Any thread in the pool can receive from any server-side connection.

The fact that the thread pool is implemented as an ASC allows a spectrum of possibilities for concurrent request handling. At one end of the spectrum each user-level thread may be directly supported by its own virtual processor (kernel thread), and at the other a single virtual processor can be used to support all user-level threads in the ASC (or the whole capsule if only one ASC per capsule is deployed). To inform the appropriate tradeoff, it can be observed that configurations involving relatively few virtual processors are more efficient in terms of context switch overhead, but is less scalable where threads perform blocking IO operations, or where multiprocessor support is available. It is also possible to dynamically change the scheduling policy in use by creating a new ASC with the appropriate policy, and then migrating the threads from the old ASC into it.

When a pool thread receives notification of the arrival of a message, it (indirectly) calls the GIOP ASP's *recv()* operation to receive the message and then performs iref-level demultiplexing and upcalls the iref's handler before eventually blocking to await receipt of further messages. Thus the architecture employs a combination of downcall and upcall structuring. If a fragmented message has being received, or if the message was erroneous, appropriate action in taken within the downcall to the ASP (i.e. storing the fragment or sending a *MessageError* message) before the thread blocks again.

4.2.8 Operation-level Demultiplexing

The following *iiop* module calls provide a service whereby skeletons can delegate operation-level demultiplexing responsibilities to the *iiop* module (alternatively, skeletons may choose to implement their own operation-level demultiplexing as is done traditionally):

```
typedef int (*Opskel)(Iref *target,
    Buffer *req, Buffer **rep);
int iiop_demuxreginterface(
    char *interface_typename,
    char *opnameset[],
    Opskel opskel[]);
int iiop_demuxreginstance(
    char *interface_typename,
    Iref *iref);
```

The *iiop_demuxreginterface()* call is used to describe an IDL interface to the *iiop* module in terms of its *opnameset* (i.e. the set of operation name strings it supports) and a congruent set of *opskels*. *Opskels* are C-functions responsible for interfacing to *one specific* method of their associated class. It is assumed that they are automatically generated by the IDL compiler. The task of each *opskel* is to unmarshal its *req* argument (assumed to contain marshalled arguments for a call to its dedicated method), upcall its dedicated method implementation on the specified target service (iref), and finally marshal any results into a reply buffer, **rep*, when this upcall has returned.

Internally, the *iiop_demuxreginterface()* call creates a *demultiplexor object* (essentially an *opnameset* $\rightarrow$ *opskel* mapping; see below) and associates this with the name *interface_typename* in preparation for a subsequent call of *iiop_demuxreginstance()*. The latter call associates a particular iref with a previously generated <*interface_typename, demultiplexor*> pair. In addition, *iiop_demuxreginstance()* links its *iref* argument to the associated demultiplexor and replaces the iref's default handler (see section 4.2.2) with a built-in generic replacement. This generic replacement handler performs the following steps:

i) it queries the associated demultiplexor object (obtained via the target iref which itself was obtained from iref-level demultiplexing) with the 'operation name' field of the incoming *Request* to obtain the address of the target *opskel*,

ii) it upcalls the target *opskel* with the request's target iref as its first parameter and the request's payload buffer as its second parameter, and,

iii) after the *opskel* has returned, it replies to the request by passing the *opskel*'s ***rep* result parameter to *iiop_send()*.

Underlying the *iiop_demux*()* services is an extensible repository of built-in *demultiplexor classes*, each of which is optimized for *opnamesets* with particular characteristics. For example, the most efficient (although not the most generally applicable) class assumes that the *n*'th character of each name in the *opnameset* is unique. The implementation here is simply an array of 256 *opskel* pointers accessed by the *n*'th byte of the name. On the other hand, the least efficient class which, however, works with any *opnameset*, performs a brute force comparison of the target operation string with each name in turn until a match is obtained. A range of alternative demultiplexors

with varying efficiency/ applicability trade-offs between these two extremes is also provided (e.g. for *opnamesets* in which the *n*th two-character sequence viewed as an integer is unique).

To determine the applicability of the various demultiplexor classes to a given *opnameset*, each demultiplexor class supports an *applicability testing method* which takes an *opnameset* as its argument and returns a boolean value indicating whether or not the demultiplexor class will work with that *opnameset*. The *iiop_demuxreginterface()* call selects an appropriate demultiplexor by placing all the demultiplexor classes in a list ranked in order of efficiency and then successively calling the applicability testing methods of successive classes until a suitable one is found. This ensures that the best available demultiplexor is applied in all cases.

5. Non-Standard Bindings

5.1 Capsule-to-iref IIOP Bindings

Capsule-to-iref IIOP bindings (see section 3.2.3) are optimized and QoS configurable variants of standard (capsule-to-capsule) IIOP bindings. The potential for optimization arises from the fact that these bindings are associated directly with a target iref rather than only indirectly via the target capsule. The client-side API for capsule-to-iref IIOP bindings is the same as that of standard IIOP bindings except that variants of the client-side *iiop_getbinding()* and *iiop_putbinding()* calls are provided:

```
int iiop_getirefbinding(
    Iref *serv, boolean exclusive);
int iiop_putirefbinding(int asap,
    Buffer *request, **reply);
```

The *iiop_getirefbinding()* and *iiop_putirefbinding()* calls work similarly to their capsule-to-capsule counterparts. The difference is that they operate on a separate binding cache which is indexed by *iref identifier* rather than by IP address/ port pairs.

Capsule-to-iref IIOP bindings essentially build a simple ‘meta-protocol’ on top of the standard IIOP binding to pass state from the client to the server and thereby improve performance. The first time *iiop_invoke()* is called on a new capsule-to-iref binding, the *iiop* module adds a special ‘marker’ to the service context field of the request. Subsequently, the server-side *iiop* module, on detecting this marker, takes some non-standard action. In particular, it may create a dedicated thread for the server side of the binding on which all subsequent invocations will be serviced, and associate this with the target iref and with the binding identifier (asap). The creation of this dedicated thread improves performance in two ways:

- by appropriately setting the priority of the thread it allows invocations on the binding to take precedence over those on other bindings, and,
- it avoids the need for iref-level demultiplexing.

The server side is at liberty to ignore the marker if it does not have sufficient resources to create a new thread; thus capsule-to-iref IIOP bindings, unlike iref-to-iref QoS bindings discussed below, do not necessarily provide any *guarantee* of enhanced service. If an attempt is made to create a capsule-to-iref binding to an ORB other than GOPI, a fully functional binding will result, although the binding will clearly not be performance enhanced (i.e. capsule-to-iref IIOP bindings are backwardly compatible with standard CORBA ORBs).

The basic scheme just described can fairly easily be adapted or extended in a large number of ways (although none of these have yet been empirically evaluated). For example, to reduce the overhead of the scheme, the dedicated thread could be per-iref rather than per-binding so that all bindings to the target iref shared the same thread. Alternatively, the server could create a *pool* of threads for the designated iref on receipt of a first marker and then add to the pool on receiving further markers from different client capsules. Further, the *Request* could include QoS parameters for the dedicated thread, or the corresponding *Reply* message could return information (e.g. on whether or not a dedicated thread had been created) to the client. A more complex extension could optimize operation-level demultiplexing by having the client side pass an operation-name-to-small-integer mapping [Gokhale,98] in its first request and then send the requisite small integer instead of the name-string in subsequent requests. The server-side *iiop* module would use the small integers and the given mapping to map more efficiently to the target operation.

A final idea would be to exploit the fact that bindings are cached on a per-iref basis to optimize the choice of the *b* parameter in the message receipt optimization of section 4.2.6. For example, in cases where the associated IDL interface employs only fixed-sized data-types, it would be possible for the IDL level to calculate and pass down at bind-time a worse case buffer size for incoming messages so that

the *msize* > *b* case would never occur[8].

5.2 Iref-to-iref QoS Bindings

Iref-to-iref QoS bindings, supported by the GOPI-core *bind* module, involve an explicit QoS-negotiated association between a specific 'proxy' iref[9] in a client capsule and a specific server iref in a remote capsule. QoS bindings may or may not involve the GIOP ASP; they are primarily useful in a GIOP context where it is desired to use GIOP over lower-level protocols other than TCP/IP[10]. Of course, because they involve negotiation, QoS bindings can only be instantiated between GOPI clients and GOPI servers; it is not possible to create a QoS binding involving a standard GIOP ORB. In addition, they carry a higher binding establishment overhead than either capsule-to-capsule or capsule-to-iref IIOP bindings.

QoS bindings bypass the *iiop* module's GIOP-specific services discussed in section 4.1.2 (with the exception of the location service) because a dedicated connection is maintained together with a dedicated per-binding thread[11] at the server side. This obviates the need for the client-side binding cache and the server-side iref-level demultiplexor with a corresponding reduction in overhead.

The client-side API for QoS bindings is as follows[12]:

```
int bind_negotiate(Iref *prox, *serv,
    CharSeq *qos, Iref *ctl);
```

In addition, the previously discussed *giop_hdr*()* family of calls are used, together with *bind_invoke()* and *bind_close()* calls which are analogous to *iiop_invoke()* and *iiop_close()* in the IIOP binding case (see section 4).

The *bind_negotiate()* call, which is the counterpart of *iiop_getbinding()* in IIOP bindings, binds a client-side 'proxy' iref *prox* to the remote service represented by iref *serv* using an ASP (which may be GIOP or any other ASP) that was associated with the *serv* iref when it was created (see section 4.2.2). The *qos* argument contains a QoS specification for the binding which is expressed in terms of the selected ASP's QoS-schema (see section 3.2.2). The *bind_negotiate()* call returns an asap together with, in *ctl*, an iref through which the binding can be controlled (in particular, to dynamically alter its QoS).

Underlying *bind_negotiate()* is the *bind* module's QoS negotiation protocol which performs the two-way QoS negotiation illustrated in figure 4. The same ASP calls that are used to establish IIOP bindings, *listen()*/ *connect()*/ *accept()*, are used by the QoS negotiation protocol. However, they are used differently: IIOP binding establishment employs *connect()* at the client side and *listen()*/ *accept()* at the server side whereas the QoS negotiation protocol calls *listen()*/ *accept()* at the client side and *connect()* at the server side (as described below).

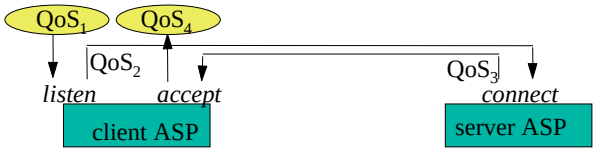

Fig. 4: The GOPI QoS negotiation protocol

In figure 4, QoS_1 is the QoS argument initially passed to the *bind_negotiate()* call. This QoS is passed to the *listen()* call of the client-side ASP which returns a new QoS, QoS_2, which may differ from QoS_1 (e.g. it may return a lower QoS if QoS_1 is too 'demanding' in some sense)[13]. The negotiation protocol then forwards QoS_2 (using an embedded IIOP invocation) to the server-side ASP where a similar mapping is performed by the *connect()* call. Subsequently, the resulting QoS_3 is returned to the client side where it is submitted to the *accept()* call of the client-side ASP for final approval or rejection of the negotiated QoS (and hence the binding itself).

Crucially, an ASP's *listen()*, *connect()* and *accept()* operations may choose (presumably on the basis of their QoS-schema parameters) to configure other ASP instances below themselves by recursively calling the corresponding connection management operations of some other ASP. This is the means by which a *stack* of ASPs is instantiated. Eventually, some ASP will terminate this recursive process by calling *listen()*/ *connect()*/ *accept()* on an underlying GOPI transport service in the *tp* module.

---

[8] This could also be done in standard IIOP bindings, albeit at the client-side only, if connections were cached on an per-iref rather than a per-port/address basis.

[9] Building on the fact that irefs are location independent, the use of an explicit proxy iref at the client side has the additional benefit that it enables *third-party binding*; i.e. bindings between arbitrarily located client and server irefs can be created from any capsule in the distributed system.

[10] It is actually possible to configure the GIOP ASP at any position in a QoS binding stack, not only at the top. The ASP can also be directly layered on top of alternate transport protocols in iref-to-iref QoS bindings.

[11] The QoS of this thread is derived (by the binding's ASP(s)) from the binding's overall QoS specification.

[12] The API given here represents only a subset of the available services. In particular, services for stream bindings are omitted. See [Coulson,99a] for details of the full API.

[13] Of course, the *listen()* call, and similarly the *connect()* and *accept()* calls, will usually do more than simply return a new QoS. In particular, it will typically perform admission tests and allocate resources for the binding. The actions taken by these calls are entirely ASP dependent.

The QoS-schema adopted by the GIOP ASP is a pair <*aspname*, *aspname_qos_spec*>. This schema allows the caller of *bind_negotiate()* to layer GIOP on top of any arbitrary ASP stack configured to any arbitrary QoS. *Aspname* is the name of the ASP type to be placed directly under GIOP (the *aspname* instance may, of course, recursively instantiate a stack of other ASPs below itself), and *aspname_qos_spec* is a QoS specification, expressed in terms of *aspname*'s schema, which is to be passed through the GIOP ASP to the *aspname* instance. Thus the GIOP ASP's QoS-schema does not contain any information that is actually interpreted by GIOP itself; all the ASP does in terms of QoS management is either:

i) configure TCP/IP connection below itself if the QoS specification is empty (as in capsule-to-capsule or capsule-to-iref bindings), or,
ii) configure an *aspname* instance below itself otherwise.

6. Performance Evaluation

6.1 Motivation and Scope

The objectives of our performance evaluation experiments were as follows:

i) to individually evaluate some of the key optimizations,
ii) to evaluate the overhead of the GOPI-core GIOP implementation relative to simple packet communication at the OS level,
iii) to evaluate the overhead of the full GOPI ORB relative to GOPI-core, and,
iv) to compare the performance of GOPI with that of other ORBs.

Some aspects of our GIOP implementation were specifically *not* subjected to a detailed evaluation. Firstly, we have not undertaken a detailed evaluation of demultiplexing overheads, because this area has already been comprehensively addressed in the literature [Gokhale,97]. Regarding operation-level demultiplexing, it is enough to say that the GOPI implementation is, as expected, indefinitely scalable in cases where the optimal demultiplexor of section 4.2.8 can be applied, and correspondingly less scalable in other cases. Similarly, iref-level demultiplexing performs as expected (e.g., we measured an approximate overhead of 5% using the basic configuration of section 6.2 when 512 objects are supported in a server capsule as opposed to the case of one object). Secondly, because their primary purpose is to provide flexibility rather than performance, we have not provided a detailed performance evaluation of the non-standard binding types. As a simple indication, we found that iref-to-iref QoS bindings yield a speedup of approximately 2% over standard IIOP bindings using the configuration of section 6.2 with 1 octet payloads.

6.2 Experimental Setup

In all cases the following experiments were executed on a single machine, an otherwise unloaded 360MHz Sun SPARC Ultra 5 with 64MB of main memory and running SunOS 5.7. The experiments all involved the same basic configuration: a client-server pair in which the client repeatedly makes round-trip calls on the server, passing an *n* octet payload in each direction. The number of calls-per-second was measured at the client, averaged over multiple runs of 10,000,000 calls. In all cases, the underlying transport was TCP/IP. All C/C++ systems, including all the ORBs considered (with the exception of Orbix 3.0; we did not have access to sources for this ORB), were compiled using GNU gcc/g++ version 2.8.1 with the -O2 optimization flag enabled.

6.3 GOPI-core Performance

This section measures the relative effects of the *Request*/ *Reply* header caching optimization and the non-blocking *recv()* optimization (objective *i*). It also relates the performance of GOPI-core-with-GIOP to 'baseline reference' levels (objective *ii*). More specifically, we compare the GIOP implementation with a C-language sockets program, a Java sockets program and a GOPI-core program configured with a simple request/ reply ASP called FRAG[14].

In these experiments, the payload buffers, which in all cases were 1 octet in size, were not touched in any way; in particular, no marshalling/ unmarshalling was performed. The C-language sockets programs were built directly on the UNIX *send()* and *recv()* system calls. The Java sockets programs accessed their sockets' *InputStream*s and *OutputStream*s using the *read(byte[], offset, length)* and *write(byte[])* methods respectively; they were written in Java 1.2.1 and executed on the Solaris 1.2.1 VM with *sunwjit*.

[14] FRAG (see [Coulson,99a]) is a request/ reply protocol that is essentially a simplified version of GIOP. It has only one message type with a 20 byte fixed-sized header. Requests and replies are distinguished by context and fixed size object identifiers are used, together with integer identifiers for operation names. Like GIOP, it assumes a reliable underlying protocol (e.g. TCP/IP). Fragmentation and endian transparency are supported but not error notification, request cancellation, object location or GIOP-like connection management. It is assumed that functions such as these will be layered on top of the basic protocol if required.

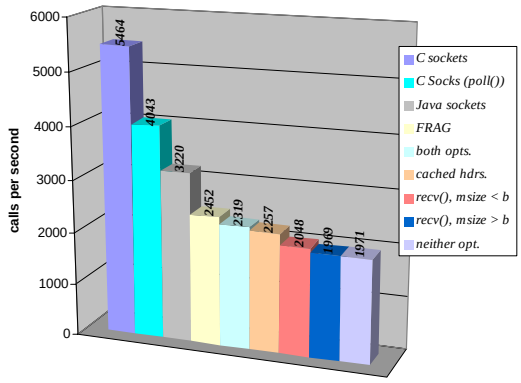

Fig. 5: GOPI-core Performance

The results of the experiments are shown in figure 5. The columns in this figure respectively illustrate the number of calls per second achieved for (from left to right):

i) the minimal C-language sockets program,
ii) a version of the above with a *poll()* call inserted before each *recv()*[15],
iii) the Java sockets program,
iv) GOPI-core/FRAG,
v) GOPI-core /GIOP with both optimizations,
vi) GOPI-core /GIOP with the header caching optimization only,
vii) GOPI-core /GIOP with the non-blocking *recv()* optimization only (in the *msize* ≤ *b* case),
viii) GOPI-core /GIOP with the non-blocking *recv()* optimization only (in the *msize* > *b* case), and,
ix) GOPI-core /GIOP with neither optimization.

It can be concluded from figure 5 that the two optimizations have a small but significant impact on GIOP performance. As expected, the non-blocking *recv()* optimization incurs a very slight liability in the pathological *msize* > *b* case. More generally, it can be seen that, despite its inherent overhead (e.g. repeated use of the *poll()* system call, buffer management, ASC thread management, inter-thread communication, ASP layer execution, GIOP header processing, iref-level demultiplexing, etc.), GOPI-core-with-GIOP delivers a throughput of around 43% (i.e. 2319 ÷ 5464) of that of the minimal C-language sockets program, 57% (i.e. 2319 ÷ 4043), of that of the *poll()* version, and around 72% (i.e. 2319 ÷ 3220) of that of the minimal Java sockets program. Furthermore, despite its higher complexity GOPI-core/ GIOP approaches the performance of the simpler FRAG protocol.

6.4 Comparative Performance

This section relates the performance of GOPI-core/GIOP to that of the full GOPI ORB (objective *iii*), and then compares the performance of the latter to that of three other prominent ORBs (objective *iv*). The selected ORBs are Washington University's TAO 1.1/ ACE 5.1, AT&T's OmniORB 2.8.0, and Iona's Orbix 3.0. TAO was selected because of its prominence in ORB research and OmniORB because it is often cited as the fastest ORB available [CC,99], [Lo,98]. Orbix was selected as it is the market leader in the commercial arena. We note, however, that Iona have more recent products to which we do not have access (e.g. Orbix 2000) which may well be faster.

The experiments employed the following IDL interface (the value SIZE was set to 1, 1024, 2048, 4096 and 8192 in successive runs to observe the effects of varying payload sizes):

```
typedef octet Data[SIZE];
interface test {
      void call(Data I, out Data O);
}
```

A 'minimal' interface was used to focus the evaluation as much as possible on the message protocol aspects of the GIOP implementations and as little as possible on the higher levels.

The results, in terms of payload size versus throughput in calls per second, are shown in figure 6. Firstly, comparing the results with those in figure 5 we observe that, for 1 byte payloads at least, GOPI's CORBA personality adds a relatively modest overhead of less than 9% ((2515 – 2319) ÷ 2319) over GOPI-core. Although our tests involved only a minimal IDL interface and GOPI ORB does not fully implement CORBA v2.2 (e.g. it omits the Portable Object Adapter–the POA, together with other higher layer CORBA services such as the dynamic invocation/ skeleton interfaces–DII/DSI), this result is interesting as it mitigates against the conventional wisdom that most ORB overhead is incurred at the higher levels such as demultiplexing and marshalling. While this is no doubt true for complex data types which require complex marshalling, our results suggest that aggressive optimization at the message protocol level can be expected yield worthwhile speedup–particularly for small/ simple messages.

[15] It is instructive to include this version of the program as a baseline reference because all ORBs need to use either *poll()* or multiple kernel threads to handle multiple connections; see also section 6.4.

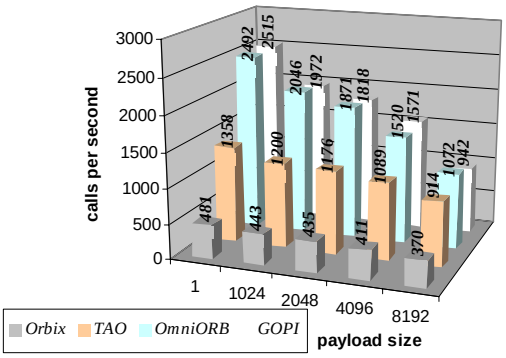

Fig. 6: General Performance Comparison

Secondly, these figures suggest that GOPI's performance compares favorably with that of other representative ORBs. In particular, it is very much on a par with OmniORB despite its relative lack of maturity and the inherent overhead of GOPI's ASC architecture (see below). OmniORB 2.8.0 is the most directly comparable ORB because, like GOPI, it does not implement the full CORBA specification.

It can also be seen that GOPI performs well in relation to TAO 1.1 and Orbix 3.0. While it is inappropriate to make direct comparisons between GOPI and these ORBs (e.g. because TAO implements both the POA and DII/DSI, and because Orbix 3.0 is an aging product and implements DII/DSI, although not the POA) the results are encouraging as far as they go.

Returning to the comparison between GOPI and OmniORB, we observe that OmniORB employs a kernel-thread-per-connection architecture. In contrast, GOPI, as part of its ASC architecture, employs a *poll()* system call before receiving each message in these tests (see section 4.2.7). This is clearly an additional overhead for GOPI in the specific context of these tests (see section 6.3 for an indication of the extent of this overhead). However, this overhead buys GOPI far greater flexibility than the kernel-thread-per-connection model, and we believe that it is a cost worth incurring. For example, it is a simple matter to configure GOPI to scale better than OmniORB where multiple clients in one capsule interact with multiple servers in another.

7. Related Work

Working groups within the OMG are currently considering ORB/ GIOP performance issues, although the work is at an early stage. In particular, the CORBA Real-Time PSIG has a working group on 'high-performance CORBA' [OMG,99b] which is investigating, among other issues, internal performance enhancement techniques and componentization. There is also a Benchmark SIG [OMG,99c] which is looking at standard ways of comparing the performance of ORB implementations.

Also from the OMG, the Real-Time CORBA specification [OMG,99e] defines various policies for real-time aspects of ORB communication such as invocation priorities, mapping bindings to multiple transports or non-multiplexed transports, and instantiating thread pools in servers to handle requests with improved predictability. However, the specification does not, of course, prescribe the implementation of these policies at the GIOP level which is the focus of our work. Many of our mechanisms would be of relevance to a Real-Time CORBA implementation. For example, ASCs could be used to implement Real-Time CORBA threadpools, and our non-multiplexed connections could form the basis of the various transport mappings.

In the academic research field, workers at Washington University, St. Louis have reported on an impressive body of work on the TAO ORB. For example, [Gokhale,98] gives a detailed performance analysis of TAO's Sunsoft-based IIOP implementation and proposes numerous optimizations. This work is complementary to the present paper in that it focuses primarily on marshalling/ unmarshalling issues rather than lower-level message handling issues. Another TAO paper [Pyarali,99] reports on a GIOP variant called GIOP-Lite which boosts performance by omitting the *magic*, *version*, *service context* and *requesting principal* fields from *Request*s. This, however, results in a non-standard, non-interoperable version of GIOP. Our approach in a situation where GIOP conformance is not required would be to use an ASP other than GIOP that is specifically optimized to the job at hand. A third TAO paper, [Gokhale,97], focuses on demultiplexing strategies. Their approach to operation-level demultiplexing is to employ the *perfect hashing* technique in the IDL compiler to automatically generate O(1) hash functions and mapping tables. While not precluding the use of this technique in GOPI, we offer the additional option of off-the-shelf demultiplexing classes below the skeleton layer as described in section 4.2.8. This approach is not as general as the perfect hashing technique although it should deliver superior best-case performance and will probably also take up less memory (i.e. our classes are written by hand, not by a code generator, and our scheme does not require a dedicated demultiplexing class for each IDL interface class).

TAO provides QoS in a GIOP environment [Schmidt,98] through three separate mechanisms:

i) the use of server threads with different priorities on which different classes of request can be serviced,
ii) the use of multiple statically configured server ports to which clients can connect, each of which is associated with a distinct QoS classification and thread priority, and,
iii) the use of per-request meta-information (e.g. a deadline for the current request) which is passed in the *Request* header's service context field.

GOPI's capsule-to-iref bindings also use the service context field for QoS related meta-information, but whereas TAO communicates information on a per-request basis, GOPI performs a once-only communication to request the dynamic establishment of a 'pseudo-connection' with a given QoS. For more general or more demanding QoS support, e.g. supporting continuous media streams, the GOPI approach is to use specialized QoS bindings, probably in conjunction with dedicated protocols (ASPs) other than GIOP. In contrast, TAO has, until recently, attempted to provide the full range of QoS behavior using the GIOP-plus-service-context technique, although [Kuhns,99] reports on more recent work in TAO that takes a more general approach to QoS provision. This is based on the 'pluggable protocol' architecture from the ORBacus commercial ORB [OOC,99] which, however, is less general in its QoS support than GOPI's ASP framework, and is not dynamically reconfigurable. A pluggable protocol framework is also just beginning to be considered by the OMG [OMG,99d].

AT&T's OmniORB [Lo,98] is another research ORB that has GIOP optimization as a prime goal. Like GOPI, OmniORB is designed to operate over multiple transports. This is achieved in OmniORB through the *strand* abstraction which encapsulates a bi-directional data connection and has a per-transport implementation. Unlike GOPI, however, OmniORB does not support general, multi-level, 'pluggable' protocol stacks. Strand instances are collected into *ropes* which encapsulate the caching of connections. OmniORB, like GOPI, can operate with a thread-per-connection model and elects not to run multiple outstanding requests over a single connection to avoid demultiplexing overhead. Unlike GOPI, however, OmniORB does not provide any QoS support beyond running over alternative transports, although the authors indicate that this issue will be investigated in their future work. Like GOPI, OmniORB is multi-threaded but its concurrency model is simpler and less flexible than that of GOPI; e.g. it does not have an equivalent of GOPI's ASC concept [Coulson,99a] and only uses native kernel threads.

Finally, there exists a significant body of research that is less directly related to our work. Our use of message header caching is related to classical work in TCP implementation [Clark,89]. A number of research ORBs exist in addition to those discussed above which, unfortunately, do not provide much detail on their GIOP implementations. These include DIMMA [Donaldson,98] and Flexinet [Hayton,97] from APM Ltd., UK, which focus on QoS provision and protocol reconfiguration respectively, Torbayou [Dang-Tran,96] and Jonathan [Dumant,98] from CNET, France Telecom which focus on open implementation techniques, and Regis [Pryce,98] from Imperial College, UK, which supports a sophisticated distributed configuration architecture.

8. Qualitative Assessment of GIOP

Although we believe GIOP to be a well-designed standard, our experience has revealed aspects of the protocol that appear to mitigate against efficient implementation. In particular, the variable sized header of most message types leads to a non-negligible overhead[16] in header parsing, and also requires separately allocated header and payload buffers and the consequent use of scatter/ gather IO. An alternative design (cf. GOPI's FRAG ASP) would have been to employ a fixed-sized *generic header* for all message types which contains only *offsets* to the various fields which are themselves held in the rest of the packet. For example, a suitable generic header may have the following fields (in addition to the currently defined fixed-sized header fields): *req_identifier*, *serv_ctx_offset*, *object_key_offset*, *op_offset*, *req_principal_offset*, *payload_offset*. The stub layer would then be responsible for marshalling the variable sized fields into the packet and informing the ASP layer of the offsets. This design would eliminate the need for header parsing and would make GOPI's *ReqState* structure redundant (i.e. the address of the header itself could be passed directly to the skeleton layer which could then directly access the required fields). It would also eliminate the need to cache multiple headers and would enable exploitation of previously developed optimizations for fixed-sized header handling [Coulson,99a].

A related issue is the use of *variable sized strings* for operation names. This leads to a non-negligible overhead in operation-level

[16] We believe that this is the main reason why GIOP is slower than the FRAG protocol.

demultiplexing (in the general case at least). It also means that it is not easy to extend IDL to use overloaded operation names (cf. C++ and Java in which multiple operations in an interface may have the same name if their arguments or return types are different). An alternative design here would have been to represent operation names as small integers. Another apparently sub-optimal aspect of the header design is the use of an octet (boolean) field in the *Request* message to specify whether or not a reply is expected. An alternative encoding of this as a bit in the flags field of the fixed-size message header would have saved four octets in the *Request* header (including the three octets used to pad the *response expected* boolean)[17]. A final header-related issue is the positioning of the *requesting principal* field at the rear of the *Request* header. This positioning obliges the *giop_hdrdefaultreq()* service to fill in the default *requesting principal* value as well as the given operation string, because the position of the former cannot be known until the length of the latter is known. This (admittedly slight) overhead could have been avoided by placing all fields that have plausible defaults (i.e. everything except the object key and operation string) towards the front of the header[18].

As mentioned above, our experience also indicates that significant efficiency gains can be achieved with negligible loss of functionality by foregoing the use of certain aspects of the standard while remaining fully GIOP compliant. In particular, we have identified the following features as being largely redundant, at least for our purposes:

- *Request multiplexing* The facility to multiplex requests on a TCP/IP connection is useful in environments in which the overhead of maintaining many connections is prohibitive (e.g. when stubs select CORBA's asynchronous method invocation (AMI) feature [OMG,98]). However, we have found that foregoing this, where possible, leads to a more streamlined implementation, at least in our implementation environment. In particular, we avoid the overhead of demultiplexing replies to the correct blocked thread at the client side, and maximize the benefits of the message receipt optimization (see section 4.2.6).
- *Request cancellation* When request multiplexing is not employed (or only rarely employed) the *CancelRequest* message becomes largely redundant and the effect of a *CancelRequest* can be inexpensively achieved simply by breaking the connection at the client side (see section 4.2.5). We therefore streamline our implementation by foregoing the sending of this message type at the client side (of course, we recognize the message at the server side to remain compatible with other ORBs, although we ignore it as permitted by the standard). On non-multiplexed connections it is also possible to avoid setting and reading the request identifier field as this is not useful for any purpose other than matching replies and cancellations to requests.
- *Fragmentation* The GIOP standard suggests that the ability to fragment messages is useful because the sender may have limited memory in which to buffer the whole of a large message (particularly in embedded and hard real-time ORBs with scarce and/or bounded memory), or because it is sometimes inconvenient for stubs and skeletons to predict the size of a message before it is actually built. However, in our workstation based environment we do not find ourselves constrained by memory limitations and can work around the message size prediction problem simply by allocating multiple buffers per message at the stub/ skeleton level, and sending them in a single GIOP message using scatter/ gather IO (thus leaving any fragmentation to the layers below). Although this streamlines the send-side, we must, of course, be prepared to expect fragmented messages arriving from other ORBs to maintain GIOP compatibility.
- *Connection Management* We have found it unnecessary to deal promptly with *CloseConnection* messages at the client side (see section 4.2.5). Rather, we believe it is sufficient to let the server side close connections at will, to detect closed connections at the client side when attempting to use them (or when a *CloseConnection* happens to be received instead of an expected *Reply* or *LocateReply*) and to recover by transparently creating new connections as required. While remaining GIOP compatible, this strategy avoids the need to dedicate a thread to waiting specifically for

[17] GIOP v1.2 improves on v1.1 in this respect as it packs additional flags into the 'response expected' octet (these are used to fine-tune the semantics of oneway messages). However, there are sufficient bits available to add these to the 'flags' field in the fixed-sized message header, so it would still have been possible to save these four octets (at the cost of leaving only two bits in the 'flags' field for future extensions).

[18] This situation has changed in GIOP v1.2 in which the 'requesting principal' field has been eliminated (due to the availability of the security service). Nevertheless, the general point still holds because v1.2 now places the 'service context' field, which has a plausible default value, at the rear of the header.

incoming *CloseConnection* messages, while incurring no appreciable reduction in functionality[19].

9. Conclusions

This paper has described the implementation of GIOP in a high performance ORB environment. The implementation is structured in a modular and extensible fashion and is integrated with the QoS configurable concurrency (ASC) and protocol (ASP) frameworks offered by the GOPI platform. It includes a number of novel implementation techniques (e.g. the header caching scheme, the single-*recv()* optimization, the tailorable operation-level demultiplexing scheme, and the use of a subset of GIOP while maintaining GIOP compatibility). It also incorporates numerous other optimizations such as connection caching, non-multiplexed connections, and scatter/ gather IO, which have been previously reported in the literature (see for example, [Pyarali,99] and [Lo,98]). Beyond this, it combines efficiency with flexible QoS support through the ASC and ASP frameworks and the non-standard binding services. We have shown that GOPI performs well in comparison to related ORBs.

In the immediate future, we plan to extend our design and implementation to take into account recent developments in the CORBA standard, especially the recently released GIOP v1.2 and the asynchronous messaging proposal [OMG,98]. In the longer term our future work will address the higher-level issues of multiplexing and demultiplexing in stubs and skeletons. We have already identified a number of possible optimizations in this area, most of which are directed at the avoidance of redundant copy operations. For example, when dealing with arrays of basic types, stubs and skeletons can unmarshal certain datatypes *in-situ* and simply pass pointers to the receiving application. Secondly, in capsule-to-iref bindings, structured datatype transfers can be optimized in cases where the sender and receiver share a common machine endian type and language environment (this can be established through the exchange of state in the service context field as discussed in section 5.1). In such cases, copy operations in skeletons can be avoided by assuming that the memory layout of the datatype in the buffer is directly usable by the receiver. It should also be possible to apply this approach to some pointer based-datatypes through pointer swizzling techniques.

[19] In GIOP v1.2, *CloseConnection* messages can be sent by the client as well as the server. This does not materially affect the argument of this section although it does imply that GOPI's client-side API must add the capability to send *CloseConnection* messages in the future to support GIOP v1.2.

References

[Blair,97] Blair, G.S., Stefani, J.B., "Open Distributed Processing and Multimedia, Addison-Wesley, 1997.

[CC,99] "The CORBA Comparison Project: Project Extension Final Report", Charles University, Prague, Czech Republic, http://nenya.ms.mff.cuni.cz/thegroup/

[Clark,89] Clark, D., Jacobson, V., Romkey, J. and Salwen, H., "An Analysis of TCP Processing Overhead", IEEE Communications, Vol 27, No 6, pp 23-29, June 1989.

[Coulson,98] Coulson, G and Clarke, M.W., "A Distributed Object Platform Infrastructure for Multimedia Applications", Computer Communications, Vol 21, No 9, pp 802-818, July 1998.

[Coulson,99a] Coulson, G., "A Configurable Multimedia Middleware Platform", IEEE Multimedia, Vol 6, No 1, pp 62-76, January - March 1999.

[Coulson,99b] Coulson, G., and Shakun Baichoo, "A Distributed Object Platform for Multimedia Applications", Proc. IEEE Multimedia Systems, Florence, Italy, ISBN 0-7695-0253-9, pp 122-126, June 1999.

[Coulson,00] Coulson, G., and Baichoo, S., "Experiences in Implementing a Distributed Object Platform for Multimedia Applications", Software Practice and Experience(30), pp 663-683, 2000.

[Dang-Tran,96] Dang Tran, F., Perebaskine, V., Stefani, J.B., Crawford, B., Kramer, A and Otway, D., "Binding and Streams: the ReTINA Approach", Proc. TINA '96, 1996.

[Donaldson,98] Donaldson, D., Faupel, M., Hayton, R., Herbert, A., Howarth, N., Kramer, A., MacMillan, I., Otway D. and Waterhouse, S., "DIMMA - A Multi-media ORB", Proc. Middleware '98, The Low Wood Hotel, Ambleside, England, ISBN 1-885233-088-0, September 1998.

[Dumant,98] Dumant, B., Horn, F., Dang-Tran, F. and Stefani, J.-B., "Jonathan: an Open Distributed Processing Environment in Java", Proc. Middleware '98, The Lake District, England, ISBN 1-885233-088-0, November 1998.

[Gokhale,97] Gokhale, A. and Schmidt, D.C., "Evaluating the Performance of Demultiplexing Strategies for Real-Time CORBA", Proc. Globecom '97, Phoenix, AZ, USA, 1997,

http://www.cs.wustl.edu/~schmidt/GLOBECOM-97.ps.gz.

[Gokhale,98] Gokhale, A. and Schmidt, D.C., "Principles for Optimizing CORBA Internet Inter-ORB Protocol Performance", Proc. HICSS '98, Hawaii, Jan 9th 1998, http://www.cs.wustl.edu/~schmidt/HICSS-97.ps.gz.

[Hayton,97] Hayton, R., "FlexiNet Open ORB Framework", APM Technical Report 2047.01.00, APM Ltd., Poseidon House, Castle Park, Cambridge, UK, 1997.

[ITU-T,95] ITU-T, ISO/IEC Recommendation X.902, International Standard 10746-2, "ODP Reference Model: Descriptive Model", January 1995.

[Kuhns,99] Kuhns, F., O'Ryan, C., Schmidt, D.C., Othman, O. and Parsons, J., "The Design and Performance of a Pluggable Protocols Framework for Object Request Broker Middleware", Proc. IFIP International Workshop on Protocols for High-Speed Networks (PfHSN '99), Salem, MA, USA, August 1999, http://www.cs.wustl.edu/~schmidt/PfHSN.ps.gz.

[Lo,98] Lo, S.L. and Pope, S., "The Implementation of a High Performance ORB over Multiple Network Transports", Proc. Middleware '98, The Lake District, England, ISBN 1-885233-088-0, pp 157-172, 1998, http://www.uk.research.att.com/omniORB/omniORBPerformance.html

[Microsoft,99] Microsoft's DCOM web page: http://windows.microsoft.com/com/tech/dcom.asp.

[OG,99] Open Group's DCE web page: http://www.opengroup.org/pubs/catalog/dz.htm

[OMG,98] CORBA Messaging Submission, http://cgi.omg.org/cgi-bin/doc?orbos/98-05-05

[OMG,99a] The Common Object Request Broker: Architecture and Specification, available at http://www.omg.org/

[OMG,99b] CORBA Real-Time PSIG High Performance WG Homepage, http://www.omg.org/homepages/realtime/working_groups/high_performance_corba.html..

[OMG,99c] CORBA Benchmark PSIG, http://www.omg.org/docs/bench/.

[OMG,99d] CORBA Telecom SIG's RFP on Extensible Transport Framework, http://www.omg.org/cgi-bin/doc?telecom/99-10-05.

[OMG,99e] CORBA Real-Time PSIG's specification of Real-Time CORBA, http://www.omg.org/cgi-bin/doc?orbos/99-02-12.

[OOC,99] Object-Oriented Concepts, "ORBacus User Manual, v3.1.2, 1999, http://www.ooc.com/ob.

[Pryce,96] Pryce, N. and Crane, S., "A Uniform Approach to Communication and Configuration in Distributed Systems", Proc. 3rd Intl. Conference of Configurable Distributed Systems, Annapolis, May 1996.

[Pyarali,99] Pyarali, I., O'Ryan, C., Schmidt, D.C., Gokhale, A., Wang, N, and Kachroo, V., "Applying Optimization Principle Patterns to Real-time ORBs", Proc. 5th USENIX Conference on O-O Technologies and Systems (COOTS '99), San Diego, USA, May 1999, http://www.cs.wustl.edu/~schmidt/COOTS-99.ps.gz.

[Schmidt,97] Schmidt, D.C., Levine, D.L. and Cleeland, C., "Architectures and Patterns for High-Performance, Real-time ORB Endsystems", to appear in Advances in Computers, Academic Press, Ed., Marvin Zelkowitz, http://www.cs.wustl.edu/~schmidt/RT-ORB.ps.gz.

[Sun,99] Sun's RMI-IIOP web page: http://java.sun.com/products/rmi-iiop/index.html.