Evaluating the Performance of Demultiplexing Strategies for Real-time CORBA

Aniruddha Gokhale and Douglas C. Schmidt

gokhale@cs.wustl.edu and schmidt@cs.wustl.edu
Department of Computer Science,
Washington University
St. Louis, MO 63130, USA.*

This paper appeared in the proceedings of GLOBECOM '97, Phoenix, AZ, November, 1997.

Abstract

Efficient and predictable demultiplexing is necessary to provide real-time support for distributed object computing applications developed with CORBA. This paper presents two contributions to the study of demultiplexing for real-time CORBA endsystems. First, we present an empirical study of four CORBA request demultiplexing strategies (linear search, perfect hashing, dynamic hashing, and active demultiplexing) for a range of target objects and operations. Second, we describe how we are using the perfect hashing and active demultiplexing strategies to develop a high-performance, real-time ORB called TAO.

Keywords: Communication software, Real-time CORBA, Demultiplexing.

1 Introduction

CORBA is a distributed object computing middleware standard defined by the Object Management Group (OMG) [14]. CORBA is designed to allow clients to invoke operations on remote objects without concern for where the object resides or what language the object is written in. In addition, CORBA shields applications from non-portable details related to the OS/hardware platform they run on and the communication protocols and networks used to interconnect distributed objects [19].

However, the performance and features of current CORBA implementations are not yet suited for hard real-time systems (*e.g.,* avionics) and constrained latency systems (*e.g.,* teleconferencing) due to overheads incurred by conventional Object Request Brokers (ORBs). Our earlier work focused on measuring and optimzing ORB presentation layer overhead to improve throughput [5, 6] and reduce latency [7]. This paper extends our earlier work by focusing on strategies for optimizing CORBA *demultiplexing* overhead.

This paper is organized as follows: Section 2 presents an overview of CORBA demultiplexing and compares our work to related research; Section 3 outlines our CORBA/ATM testbed and describes the methodology for our demultiplexing experiments; Section 4 examines the results of our experiments and analyzes the overhead associated with each demultiplexing strategy; and Section 5 presents concluding remarks.

2 Overview of CORBA Demultiplexing

2.1 Conventional CORBA Demultiplexing Architectures

A CORBA request header contains the identity of its remote object implementation (which is called a *servant* by the CORBA specification [14]) and its intended remote operation. A servant is uniquely identified by an *object key* and an *operation name.* An object key is represented as an IDL `sequence`, which is a single dimensional dynamic array of bytes; an operation name is represented as a string.

The Object Adapter is the component in the CORBA architecture that associates a servant with the ORB, demultiplexes incoming requests to the servant, and dispatches the appropriate operation of that servant. While current CORBA implementations typically provide a single Object Adapter per ORB, recent ORBOS portability enhancements [14] define the Portable Object Adapter (POA) to support multiple Object Adapters per ORB.

The demultiplexing strategy used by an ORB can impact performance significantly. Conventional ORBs demultiplex client requests to the appropriate operation of the servant using the following steps shown in Figure 1.

• **Steps 1 and 2:** The OS protocol stack demultiplexes the incoming client request multiple times, *e.g.,* through the data link, network, and transport layers up to the user/kernel boundary and the ORB core;

• **Steps 3, 4, and 5:** The ORB core uses the addressing information in the client's object key to locate the appropriate Object Adapter, servant, and the skeleton of the target IDL operation;

*This work was supported in part by NSF grant NCR-9628218, Sprint, and Boeing.

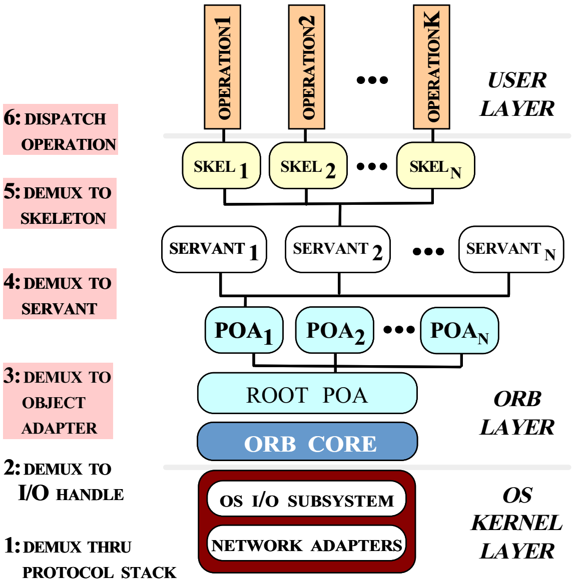

Figure 1: Layered CORBA Request Demultiplexing

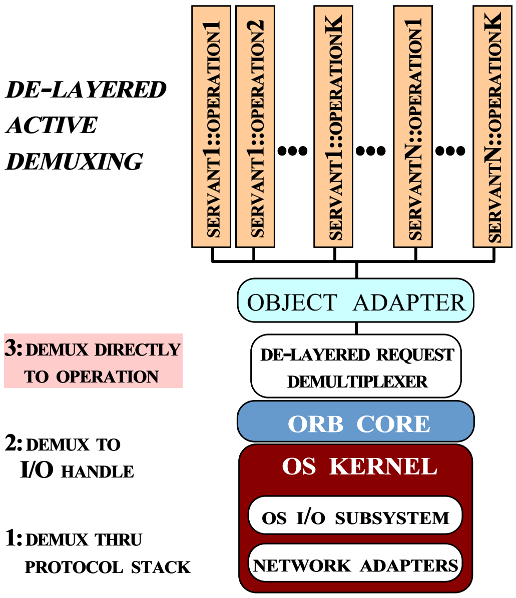

Figure 2: De-layered CORBA Request Demultiplexing

- **Step 6:** The IDL skeleton locates the appropriate operation, demarshals the request buffer into operation parameters, and performs the operation upcall.

Demultiplexing client requests through all these layers is expensive, particularly when a large number of operations appear in an IDL interface and/or a large number of servants are managed by an ORB. Layered demultiplexing is particularly inappropriate for real-time applications [18] because it increases latency by increasing the number of times that internal tables must be searched while incoming client requests traverse various protocol processing layers. In addition, layered demultiplexing can cause priority inversions because servant-level QoS information is inaccessible to the lowest level device drivers and protocol stacks in the I/O subsystem of an ORB endsystem.

Conventional implementations of CORBA incur significant demultiplexing overhead. In particular, [5, 7] show that ~17% of the total server processing time is spent demultiplexing requests. Unless this overhead is reduced and demultiplexing is performed predictably, ORBs cannot provide real-time quality of service guarantees to applications.

2.2 Design of a Real-time Object Adapter for TAO

TAO is a high performance, real-time ORB developed at Washington University [17]. It runs on a range of OS platforms that support real-time features including VxWorks, Solaris 2.x, and Windows NT. TAO provides a highly optimized version of SunSoft's implementation of the CORBA Internet Inter-ORB Protocol (IIOP)[8].[1]

TAO's Object Adapter is designed to minimize overhead via delayered demultiplexing [18] shown in Figure 2. This approach maps client requests directly to servant/operation tuples that perform application-level upcalls. The result is $O(1)$ performance for the average- and worst-cases.

[1] SunSoft IIOP is freely available from the OMG web site at URL ftp://ftp.omg.org and the TAO ORB is available at http://www.cs.wustl.edu/~schmidt/TAO.html.

Figure 3 illustrates the components in the CORBA architecture and the various demultiplexing strategies supported by TAO. TAO's flexible design allows different demultiplexing strategies [16] to be plugged into its Object Adapter. Section 4 presents the results of experiments using the following four demultiplexing strategies: (A) linear search, (B) perfect hashing, (C) dynamic hashing, and (D) de-layered active demultiplexing shown in Figure 3:

Linear search: The linear search demultiplexing strategy is a two-step layered demultiplexing strategy (shown in Figure 3(A)). In the first step, the Object Adapter uses the object key to linearly search through the active object map[2] to locate the right object and its skeleton (each entry in an active object map maintains a pointer to its associated skeleton). The skeleton maintains a table of operations defined by the IDL interface. In the second step, the Object Adapter uses the operation name to linearly search the operation table of the associated skeleton to locate the appropriate operation and invoke an upcall on it.

Linear search is known to be expensive and non-scalable. We include it in our experiments for two reasons: (1) to provide an upper bound on the worst-case performance, and (2) to contrast our optimizing demultiplexing strategies with strategies used in existing ORBs (such as Orbix) that use linear search for their operation demultiplexing.

Perfect hashing: The perfect hashing strategy is also a two-step layered demultiplexing strategy (shown in Figure 3(B)). In contrast to linear search, the perfect hashing strategy uses an automatically-generated perfect hashing function to locate the servant. A second perfect hashing function is then used to locate the operation. Both servant and operation lookup take constant time.

Perfect hashing is applicable when the keys to be hashed are known *a priori*. In many hard real-time systems (such as avionic control systems [9]), the objects and operations can be configured

[2] The active object map [14] associates object keys to servants maintained by an Object Adapter.

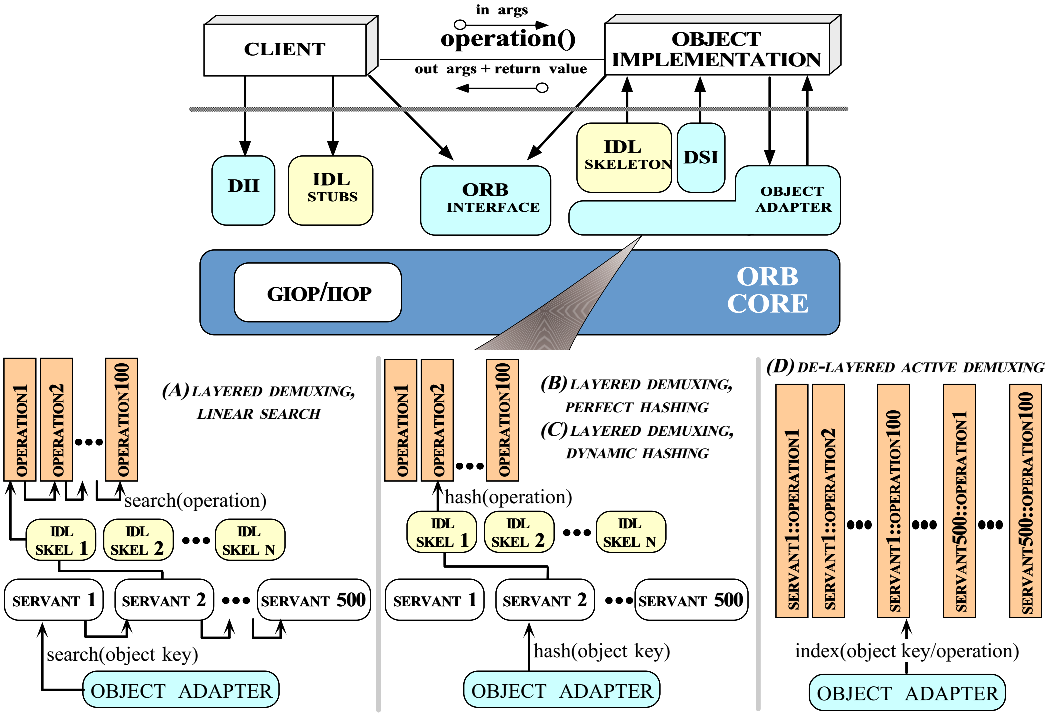

Figure 3: Alternative Demultiplexing Strategies in TAO

statically. In this scenario, it is possible to use perfect hashing to hash the object and operations. For our experiment, we used the GNU `gperf` [15] tool to generate perfect hash functions for object keys and operation names.

The following is a code fragment from the GNU `gperf` generated hash function for 500 object keys used in our experiments:

```
class Object_Hash
{
  // ...
  static u_int hash (const char *str, int len);
};

u_int
Object_Hash::hash (register const char *str,
                            register int len)
{
  static const u_short asso_values[] =
    {
     // all values not shown here
     1032, 1032, 1032, 1032, 1032, 1032, 1032,
      100,  105,  130,   20,  100,  395,  435,
      505,  330,  475,   45,  365,  180,  390,
      440,  160,  125, 1032, 1032, 1032, 1032,
    };
  return len + asso_values[str[len - 1]]
           + asso_values[str[0]];
}
```

The code above works as follows: upon receiving a client request, the Object Adapter retrieves the object key. It uses the object key to obtain a handle to the active object map by using the perfect `hash` function shown above. The `hash` function uses an automatically generated active object map (`asso_values`) to return a unique hash value for each object key.

Dynamic hashing: The dynamic hashing strategy is also a two-step layered demultiplexing strategy (shown in Figure 3(C)). In contrast to perfect hashing, which has $O(1)$ worst-case behavior and low constant overhead, dynamic hashing has higher overhead and $O(n^2)$ worse-case behavior. In particular, two or more keys may dynamically hash to the same bucket. These collisions are resolved using linear search, which can yield poor worse-case performance. The primary benefit of dynamic hashing is that it can be used when the object keys are not known *a priori*. In order to minimize collisions, the object and the operation hash tables contained twice as many array elements as the number of servants and operations, respectively.

De-layered active demultiplexing: The fourth demultiplexing strategy is called *de-layered active demultiplexing* (shown in Figure 3(D)). In this strategy, the client includes a handle to the object in the active object map and the operation table in the CORBA request header. This handle is configured into the client when the target object reference is registered with a Naming service or Trading service. On the receiving side, the Object Adapter uses the handle supplied in the CORBA request header to locate the object and its associated operation in a single step.[3]

2.3 Related Work

Related work on demultiplexing focuses largely on the lower layers of the protocol stack (*i.e.*, the transport layer and below) as opposed to the CORBA middleware. For instance, [18, 4, 2] study demultiplexing issues in communication systems and show how layered demultiplexing is not suitable for applications that require real-time quality of service guarantees.

Packet filters are a mechanism for efficiently demultiplexing incoming packets to application endpoints [13]. A number of

[3]Detailed description on the four demultiplexing strategies is available at URL `http://www.cs.wustl.edu/~schmidt/GLOBECOM-97.ps.gz`

schemes to implement fast and efficient packet filters are available. These include the BSD Packet Filter (BPF) [12], the Mach Packet Filter (MPF) [20], PathFinder [1], demultiplexing based on automatic parsing [11], and the Dynamic Packet Filter (DPF) [3].

Most existing demultiplexing strategies are implemented within the OS kernel. However, to optimally reduce ORB endsystem demultiplexing overhead requires a vertically integrated architecture that extends from the OS kernel to the application servants. Since our ORB is currently implemented in user-space, however, this paper focuses on minimizing the demultiplexing overhead in steps 3, 4, 5, and 6 (which are shaded in Figure 1).

3 CORBA/ATM Testbed and Experimental Methods

3.1 Hardware and Software Platforms

The experiments in this section were conducted using a FORE systems ASX-1000 ATM switch connected to two dual-processor UltraSPARC-2s running SunOS 5.5.1. The ASX-1000 is a 96 Port, OC12 622 Mbs/port switch. Each UltraSPARC-2 contains two 168 MHz CPUs with a 1 Megabyte cache per-CPU. The SunOS 5.5.1 TCP/IP protocol stack is implemented using the STREAMS communication framework.

In addition, each UltraSPARC-2 has 256 Mbytes of RAM and an ENI-155s-MF ATM adapter card, which supports 155 Megabits per-sec (Mbps) SONET multimode fiber. The Maximum Transmission Unit (MTU) on the ENI ATM adapter is 9,180 bytes. Each ENI card has 512 Kbytes of on-board memory. A maximum of 32 Kbytes is allotted per ATM virtual circuit connection for receiving and transmitting frames (for a total of 64 K). This allows up to eight switched virtual connections per card. This hardware platform is shown in Figure 4.

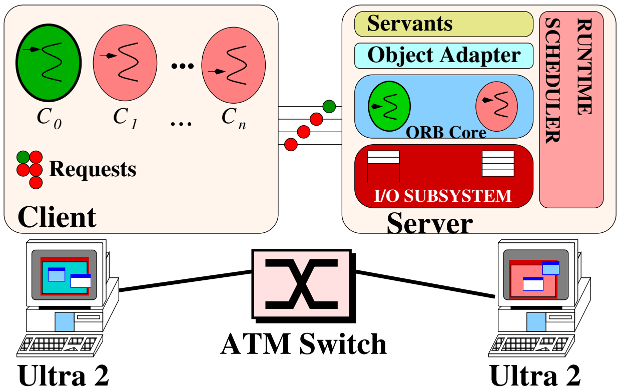

Figure 4: Hardware for the CORBA/ATM Testbed

3.2 Parameter Settings

Our earlier studies [5, 7] of CORBA performance over ATM demonstrate the performance impact of parameters such as the number of servants on an endsystem (*e.g.,* a server), and interfaces with large number of methods. Therefore, our benchmarks systematically varied these parameters for each experiment as follows:

- **Number of servants:** Increasing the number of objects on the server increases the demultiplexing effort required to dispatch the incoming request to the appropriate object. To pinpoint this demultiplexing overhead and to evaluate the efficiency of different demultiplexing strategies, we benchmarked a range of objects (1, 100, 200, 300, 400, and 500) on the server.

- **Number of operations defined by the interface:** In addition to the number of objects, demutiplexing overhead increases with the number of operations defined in an interface. To measure this demultiplexing overhead, our experiments defined a range of operations (1, 10, and 100) in the IDL interface. Since our experiments measured the overhead of demultiplexing, these operations defined no parameters, thereby eliminating the overhead of presentation layer conversions.

3.3 Request Invocation Strategies

Our experiments used two different invocation strategies for invoking different operations on the server objects. The two invocation strategies are:

- **Random request invocation strategy:** In this case, the client makes a request on a randomly chosen object reference for a randomly chosen operation. This strategy is useful to test the efficiency of the hashing-based strategy. In addition, it measures the average performance of the linear search based strategy. The algorithm for randomly sending client requests is shown below.

```
for (int i = 0; i < NUM_OBJECTS; i++) {
  for (int j = 0; j < NUM_OPERATIONS; j++) {
     choose an object at random from
        the set [0, NUM_OBJECTS - 1];
     choose an operation at random from
        the set [0, NUM_OPERATIONS - 1];
     invoke the operation on that object;
  }
}
```

- **Worst-case request invocation:** In this case, we choose the last operation of the last object. This strategy elicits the worst-case performance of the linear search strategy. The algorithm for sending the worse-case client requests is shown below:

```
for (int i = 0; i < NUM_OBJECTS; i++) {
  for (int j = 0; j < NUM_OPERATIONS; j++) {
     invoke the last operation on the
       last object
  }
}
```

3.4 Profiling Tools

The `gethrtime` system call was used to determine demultiplexing overhead by computing detailed timing measurements. This system call uses the SunOS 5.5 high-resolution timer, which expresses time in nanoseconds from an arbitrary time in the past. The time returned by `gethrtime` is very accurate since it does not drift.

The profile information for the empirical analysis was obtained using the `Quantify` [10] performance measurement tool. `Quantify` analyzes performance bottlenecks and identifies sections of code that dominate execution time. Unlike traditional sampling-based profilers (such as the UNIX `gprof` tool), `Quantify` reports results without including its own overhead. In addition, `Quantify` measures the overhead of system calls and third-party libraries without requiring access to source code.

All data is recorded in terms of machine instruction cycles and converted to elapsed times according to the clock rate of the machine. The collected data reflect the cost of the original program's instructions and automatically exclude any `Quantify` counting overhead.

4 Demultiplexing Performance Results

This section presents our experimental results that measure the overhead of the four demultiplexing strategies described in Section 2.2. Section 4.1 presents the blackbox results of our experiments. Section 4.2 provides detailed whitebox analysis of the blackbox results. Section 4.3 evaluates the pros and cons of each demultiplexing strategy.

4.1 Performance Results for Demultiplexing Strategies

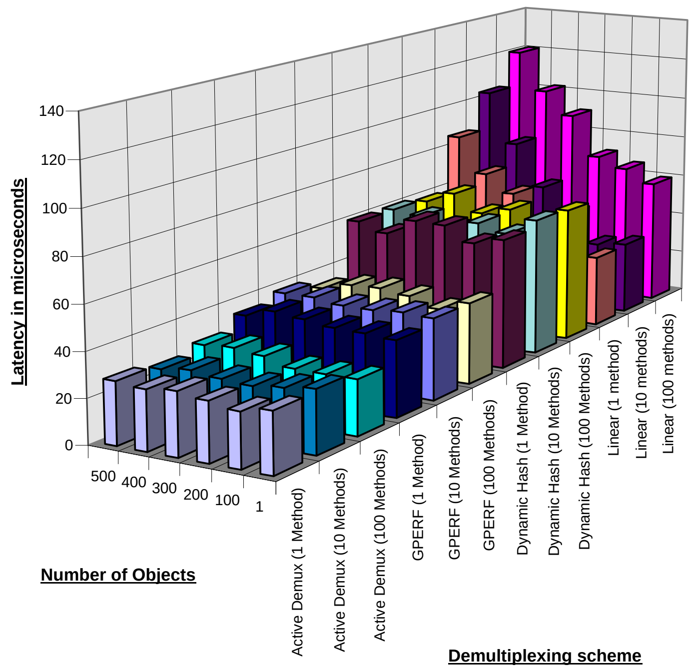

Figure 5: Demultiplexing Overhead for the Random Invocation Strategy

Figures 5 and 6 illustrate the performance of the four demultiplexing strategies for the random and worst-case invocation strategies, respectively. These figures reveal that in both cases, the de-layered active demultiplexing and perfect hash-based demultiplexing strategies substantially outperform the linear-search strategy and the dynamic hashing strategy. Moreover, the worst-case performance overhead of the linear-search strategy for 500 objects and 100 operations is $\sim$1.87 times greater than random invocation, which illustrates the non-scalability of linear search as a demultiplexing strategy.

In addition, the figures reveal that both the active demultiplexing and perfect hash-based demultiplexing perform quite efficiently and predictably regardless of the invocation strategies. The de-layered active demultiplexing strategy performs slightly better than the perfect hash-based strategy for both invocation strategies. Section 4.2 explains the reasons for these results.

4.2 Detailed Analysis of Demultiplexing Overhead

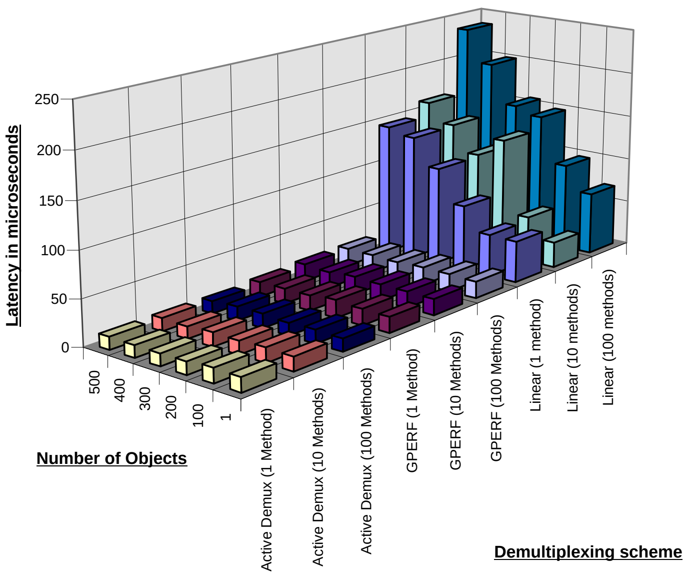

Figure 6: Demultiplexing Overhead for the Worst-case Invocation Strategy

This section presents the results of our whitebox profiling to illustrate the overhead of each demultiplexing strategy shown in Section 4.1. We explain the `Quantify` results from invoking 100 operations on 500 objects using the worst-case invocation strategy in Table 1.

The `dispatch` method used for our experiments is shown below:

```
int
CORBA_BOA::dispatch (CORBA_OctetSequence key,
                     CORBA_ServerRequest& req,
                     CORBA_Object_ptr obj)
{
   skeleton *skel;
   CORBA_Object_ptr obj;
   CORBA_String opname;

   // Find the object ptr corresponding
   //  to the key in the object table.
   // Use one of the 4 demux strategies
   if (this->find (key, obj) {
```

Strategy	Analysis			
	Name	Time in msec	Called	%
Active demux	OA::find	280.45	50,000	9.14
	skeleton	249.94	50,000	8.14
	Object::find	36.71	50,000	1.20
Perfect hash	OA::find	346.42	50,000	23.73
	skeleton	253.62	50,000	17.37
	Object::find	78.67	50,000	5.39
Dynamic hash	OA::find	670.56	50,000	34.13
	Object::find	265.39	50,000	13.51
	skeleton	253.62	50,000	12.91
Linear search	OA::find	12,122.73	50,000	72.46
	Object::find	3,617.48	50,000	21.62
	skeleton	249.65	50,000	1.49

Table 1: Analysis of Demultiplexing Overhead

```
      opname = req.opname ();
      // Now find the skeleton corresponding
      //  to the operation name.
      if (obj->find (opname, skel) {
        // invoke the skeleton
      }
    }
}
```

Only the overhead incurred by the `CORBA::OA::dispatch` method and its descendants are shown in Table 1. The primary descendants of the `dispatch` method include the following:

- `OA::find` – which locates a servant corresponding to an object key in the object table maintained by the `OA`;
- `Object::find` – which locates a skeleton corresponding to the operation name in the operation table in the servant;
- `The skeleton` – which parses any arguments and makes the final upcall on the target servant.

Column `Name` identifies the name of the high cost descendants. The execution time is shown under column `Time in msec`. The `Called` column indicates the number of times the method was invoked and `%` indicates the percentage of the total execution time incurred by this method and its descendants.

Table 1 reveals that the linear-search based strategy spends a substantial portion of its time in the `OA::find` and `Object::find`. In turn, they perform string comparisons on the object keys and operation names, respectively. In contrast, both the hashing-based and active demultiplexing strategies incur no measurable overhead to locate the object and the associated operation. The dynamic hashing scheme involves fewer string comparisons compared to the linear search strategy and hence it performs better.

The `OA::dispatch` method and its descendants account for ~20% of the total execution time for the *De-layered Active Demultiplexing* strategy, ~50% for the *Perfect hash* strategy, ~63% for the *Dynamic hash* strategy, and ~95% for the *Linear search* strategy. This explains why the de-layered active demultiplexing strategy outperforms the rest of the strategies.

4.3 Analysis of the Demultiplexing Strategies

The performance results and analysis presented in Sections 4.1 and 4.2 reveal that to provide low-latency and predictable real-time support, a CORBA Object Adapter must use demultiplexing strategies based on active demultiplexing or perfect hashing rather than strategies such as linear-search (which does not scale) and dynamic hashing (which has high overhead).

The perfect hashing strategy is primarily applicable when the object keys are known *a priori*. The number of operations are always known *a priori* since they are defined in an IDL interface. Thus, an IDL compiler can generate stubs and skeletons that use perfect hashing for operation lookup. However, objects implementing an interface can be created dynamically. In this case, the perfect hashing strategy cannot generally be used for object lookup.[4] In this situation, more dynamic forms of hashing can be used as long as they provide predictable collision resolution strategies. In many hard real-time environments it is possible to configure the system *a priori*. In this situation, however, perfect hashing-based demultiplexing can be used.

Our results show that de-layered active demultiplexing outperforms the other demultiplexing strategies. However, it requires the client to possess a handle for each object and its associated operations in the active object map and operation tables, respectively. Therefore, active demultiplexing requires either (1) preconfiguring the client with this knowledge or (2) defining a protocol for dynamically managing handles to add and remove objects correctly and securely.[5]

For hard real-time systems, this preconfiguration is typically feasible and beneficial. For this reason, we are using the perfect hashing demultiplexing strategy in the TAO ORB we are building for real-time avionics applications [17, 9].

5 Concluding Remarks

An important class of applications (such as avionics, virtual reality, and telecommunication systems) require scalable, low latency communication. Our earlier work [5, 7] shows why latency-sensitive applications that utilize many servants and large interfaces are not yet supported efficiently by contemporary CORBA implementations due to demultiplexing overhead, presentation layer conversions, data copying, and many layers of virtual method calls. On low-speed networks this overhead is often masked. On high-speed networks, this overhead becomes a significant factor limiting communication performance and ultimately limiting adoption of CORBA by developers.

The results presented in this paper compare four demultiplexing strategies. The object and operation lookup used by these strategies are based on linear-search, perfect hashing, dynamic hashing, and de-layered active demultiplexing. Our results reveal that the de-layered active demultiplexing strategy provides the lowest latency and most predictability amongst the three strategies studied. We have currently implemented the de-layered active

[4] It is possible to add new objects at run-time using dynamic linking, though this is generally disparaged in hard real-time environments.

[5] We assume that the security implications of using active demultiplexing are addressed via the CORBA security service.

demultiplexing strategy in TAO, which is our high performance, real-time ORB [17].

References

[1] Mary L. Bailey, Burra Gopal, Prasenjit Sarkar, Michael A. Pagels, and Larry L. Peterson. Pathfinder: A pattern-based packet classifier. In *Proceedings of the 1^{st} Symposium on Operating System Design and Implementation*. USENIX Association, November 1994.

[2] Zubin D. Dittia, Guru M. Parulkar, and Jr. Jerome R. Cox. The APIC Approach to High Performance Network Interface Design: Protected DMA and Other Techniques. In *Proceedings of INFOCOM '97*, Kobe, Japan, April 1997. IEEE.

[3] Dawson R. Engler and M. Frans Kaashoek. DPF: Fast, Flexible Message Demultiplexing using Dynamic Code Generation. In *Proceedings of ACM SIGCOMM '96 Conference in Computer Communication Review*, pages 53–59, Stanford University, California, USA, August 1996. ACM Press.

[4] David C. Feldmeier. Multiplexing Issues in Communications System Design. In *Proceedings of the Symposium on Communications Architectures and Protocols (SIGCOMM)*, pages 209–219, Philadelphia, PA, September 1990. ACM.

[5] Aniruddha Gokhale and Douglas C. Schmidt. Measuring the Performance of Communication Middleware on High-Speed Networks. In *Proceedings of SIGCOMM '96*, pages 306–317, Stanford, CA, August 1996. ACM.

[6] Aniruddha Gokhale and Douglas C. Schmidt. The Performance of the CORBA Dynamic Invocation Interface and Dynamic Skeleton Interface over High-Speed ATM Networks. In *Proceedings of GLOBECOM '96*, pages 50–56, London, England, November 1996. IEEE.

[7] Aniruddha Gokhale and Douglas C. Schmidt. Evaluating Latency and Scalability of CORBA Over High-Speed ATM Networks. In *Proceedings of the International Conference on Distributed Computing Systems*, Baltimore, Maryland, May 1997. IEEE.

[8] Aniruddha Gokhale and Douglas C. Schmidt. Principles for Optimizing CORBA Internet Inter-ORB Protocol Performance. In *Hawaiian International Conference on System Sciences*, January 1998.

[9] Timothy H. Harrison, David L. Levine, and Douglas C. Schmidt. The Design and Performance of a Real-time CORBA Event Service. In *Proceedings of OOPSLA '97*, Atlanta, GA, October 1997. ACM.

[10] PureAtria Software Inc. *Quantify User's Guide*. PureAtria Software Inc., 1996.

[11] Mahesh Jayaram and Ron Cytron. Efficient Demultiplexing of Network Packets by Automatic Parsing. In *Proceedings of the Workshop on Compiler Support for System Software (WCSSS 96)*, University of Arizona, Tucson, AZ, February 1996.

[12] Steven McCanne and Van Jacobson. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *Proceedings of the Winter USENIX Conference*, pages 259–270, San Diego, CA, January 1993.

[13] Jeffrey C. Mogul, Richard F. Rashid, and Michal J. Accetta. The Packet Filter: an Efficient Mechanism for User-level Network Code. In *Proceedings of the 11^{th} Symposium on Operating System Principles (SOSP)*, November 1987.

[14] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, 2.2 edition, February 1998.

[15] Douglas C. Schmidt. GPERF: A Perfect Hash Function Generator. In *Proceedings of the 2^{nd} C++ Conference*, pages 87–102, San Francisco, California, April 1990. USENIX.

[16] Douglas C. Schmidt and Chris Cleeland. Applying Patterns to Develop Extensible ORB Middleware. *IEEE Communications Magazine*, to appear 1999.

[17] Douglas C. Schmidt, David L. Levine, and Sumedh Mungee. The Design and Performance of Real-Time Object Request Brokers. *Computer Communications*, 21(4):294–324, April 1998.

[18] David L. Tennenhouse. Layered Multiplexing Considered Harmful. In *Proceedings of the 1^{st} International Workshop on High-Speed Networks*, May 1989.

[19] Steve Vinoski. CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments. *IEEE Communications Magazine*, 14(2), February 1997.

[20] M. Yuhara, B. Bershad, C. Maeda, and E. Moss. Efficient Packet Demultiplexing for Multiple Endpoints and Large Messages. In *Proceedings of the Winter Usenix Conference*, January 1994.