Working Together in Harmony — An Implementation of the CORBA Object Query Service and its Evaluation

Uwe Röhm Klemens Böhm
Swiss Federal Institute of Technology
ETH Zentrum, 8092 Zurich, Switzerland
{roehm, boehm}@inf.ethz.ch

Abstract

The CORBA standard, together with its service specifications, has gained considerable attention in recent years. The CORBA Object Query Service allows for declarative access to heterogeneous storage systems. We have come up with an implementation of this service called Harmony. The objective of this article is to provide a detailed description and quantitative assessment of Harmony. Its main technical characteristics are data-flow evaluation, bulk transfer, and intra-query parallelism. To carry out the evaluation, we have classified data exchange between components of applications in several dimensions: one is to distinguish between point-, context- and bulk data access. We have compared Harmony with (1) data access through application-specific CORBA objects, and (2) conventional client/server communication, i.e., Embedded SQL. Our results show that Harmony performs much better than Alternative 1 for bulk data access. Besides that, due to the features mentioned above, Harmony performs approximately as well as conventional client/server communication mechanisms.

1. Introduction

CORBA [17] is a middleware standard that has gained much attention in the recent past, also from industry. It specifies a communication infrastructure for arbitrary heterogeneous components of distributed systems. Furthermore, it defines a set of basic system services and standard components. One service that is of particular interest from an information system point of view is the *Object Query Service (OQS)*. A well-designed implementation of this service provides transparent declarative access to heterogeneous storage systems.

Taking the importance of CORBA and the OQS specification for granted, we for our part have designed and realized an implementation of this service called *Harmony*. To the best of our knowledge, other implementations of the CORBA OQS do not exist (cf. [6]). As we explain in Section 3, an efficient implementation of this service is not straightforward. To come up with a realization of the query service whose performance is competitive, we have deployed classical database concepts, notably dataflow evaluation, bulk transfer and intra-query parallelism: internally, Harmony evaluates a query in a dataflow manner according to an execution plan. Each operation of the plan corresponds to a *QueryableCollection* object instantiated before evaluating the query. These objects actually implement the query algebra operators of Harmony. In addition to the usual operators, the query algebra includes *meta operators*. Meta operators do not change the data stream, but perform some control function. This allows us to introduce bulk transfer and intra-query parallelism, as we will explain in Subsection 4.1. A platform-independent realization. i.e., for arbitrary ORBs, may be cumbersome. Hence, we identify features that some commercially available ORBs have, and that allow to implement bulk transfer and intra-query parallelism elegantly. We then integrate different storage systems through *wrappers*. Wrappers implement another meta operator.

To evaluate Harmony, we compare it to the following alternatives for data access:

1. Application-specific CORBA objects, and
2. direct, low-level data access via Embedded SQL/C.

These approaches merely serve as orientation points regarding Harmony's performance; they are not fully compatible alternatives. Alternative 1 is the usual way to access data in a CORBA-based environment: application-specific CORBA objects perform a specific task and encapsulate persistent data whose type is static. This alternative does not allow for declarative data access. Alternative 2, as opposed to Harmony, does not provide transparent access to heterogeneous repositories.

To come up with an evaluation that is as differentiated as possible, we classify data access in business environments in several dimensions. One of the classifications partly reflects the amount of data being transferred. Its categories are *Point Data Access* to single data items, *Context Data Access* to a number of items, and *Bulk Data Access* to large amount of data. For evaluation, access to bulk data seems to be the most interesting category from our perspective. Consequently, we use a fragment of the TPC/D database schema, together with its data, as experimental platform. Next to varying the quantity of data, we also investigate the impact of the quality of the communication network. Finally, we assess the ease of integration of new data sources. The main results of our experiments are the following ones: with a naive implementation of a CORBA-based solution, the CORBA-inherent overhead for access to bulk data is significant. But with asynchronous query evaluation, as is the case with our implementation, Harmony outperforms application-specific CORBA objects for context and bulk data access, both with respect to execution times and resource consumption. Furthermore, Harmony is superior with regard to the application-specific implementation effort required. Finally, its runtime overhead compared to direct client/server communication with ESQL/C is pleasingly low, coming out at about 30% in the average case.

Our work on Harmony falls in the area of query execution in a distributed, heterogeneous environment. Query optimization, multi-database issues, as well as a comparison with OLE/COM are not part of this current work. The remainder of this paper is organized as follows: the next section further describes the relevance of CORBA and its Object Query Service in mostly non-technical terms, and classifies data exchange in business environments. The subsequent two sections give a technical overview of the relevant aspects of CORBA and the implementation of Harmony. While Section 3 summarizes the techniques available for data source integration into a CORBA environment, Section 4 concentrates on the Harmony OQS and the relevant aspects of query processing. Section 5 contains the evaluation of Harmony for the different usage scenarios. Section 6 compares the development complexity required by the three alternative access techniques. The article ends with an overview of relevant work and concluding remarks. An extended version is available from the authors [19].

2. Motivation

Why CORBA? Modern software artifacts consist of components that cooperate. A middleware layer separates the front ends of an application from its back ends. This gives rise to location and implementation transparency. In other words, changes to component implementations or component relocation do not affect the clients. This reduces the complexity of applications and promotes interoperability. CORBA [17] is a standard specification of such a middleware layer. In a CORBA environment, all interactions between components pass through the *Object Request Broker (ORB)*. In a nutshell, an ORB provides an object-oriented remote procedure call mechanism with automatic conversion between different operating systems and programming languages. The CORBA standard also defines a set of system services as building blocks for complex applications. One such service is the Object Query Service (OQS). With regard to data exchanges, this service is of particular interest, as it allows for declarative queries over heterogeneous storage systems. Queries can be formulated and submitted at runtime.

Classifying data access in business environments. To provide further insight into the nature of data exchange in such distributed, heterogeneous environments, we classify data access in business environments in several dimensions. Another reason for introducing this classification is for evaluation purposes in Section 5: we establish a framework for evaluating Harmony. As a first dimension, we distinguish the following kinds of data access:

- Point Data Access
  Subject of interest is a single data item, e.g., a customer wants to know the price of a certain product.

- Context Data Access
  Some data access operations do not only refer to a single data item, but a small number of items. Imagine for example a salesperson who wants to know the recent orders of a certain customer.

- Bulk Data Access
  Access to large amounts of data is required if one surveys the data, e.g., retrieve an address list of all customers, or runs decision support applications.

The dividing line between context and bulk access scenarios is subject to interpretation. But it is important to make the distinction. Context access is more common in a distributed environment than bulk data access. However, bulk data access is still necessary. These three categories correspond to different numbers of data items accessed. But this quantity alone does not specify the amount of data transferred. If we are also interested in quality constraints, the above categorization does not suffice. Conformant to OMG's classification of CORBA services and facilities [16], another categorization of data exchange in commercial environments is along a second dimension:

- Horizontal Market
  Data access in this category is independent of a specific business area, e.g., access to customer information.

- Vertical Market
  Data access may be specific to a certain market. In this case, the data may have certain characteristics, e.g.,

complex objects and multimedia data. Furthermore, data access is subject to further constraints, e.g., high performance or quality of service.

Finally, another classification of data access in heterogeneous environments corresponds to the kind of data sources. Typically, data is not only stored in a (single) central database, but also in application-specific repositories, e.g., CAD systems, image repositories, SAP, or files.

3. Data Access in a CORBA Environment

The usual way of integrating data sources into a CORBA environment is by wrapping them with application-specific CORBA objects. Data sources are accessible through the public methods of these objects. Note that one can directly access only data for which the application programmer has provided access methods. In Subsection 3.2, we provide the reader with more details on this approach, because it is used for performance comparison with Harmony in Section 5.

Bulk access to data via CORBA, if implemented in a straightforward way, may introduce significant overhead, compared to "home-grown" communication mechanisms, as this section explains. There is overhead inherent to any CORBA-based solution (Subsection 3.1), overhead specific to application-specific CORBA objects (Subsection 3.2) or overhead occurring with a straightforward implementation of the declarative query service (Subsection 3.3).

3.1. Data Exchange via CORBA Methods

The communication mechanisms of CORBA are as follows. One has to describe each CORBA object with a public interface description in CORBA's *Interface Definition Language (IDL)*. From this description an IDL compiler generates code for the client and the server side. The client part consists of stub code. Stub code provides so-called *proxy objects* in the client address space. Proxy objects just route all method calls through the *Object Request Broker* to the actual implementation somewhere in the system. As CORBA is an international standard, various vendors are shipping their own ORB implementation. With some of them, e.g., with Orbix [14] we have used, the programmer can change the ORB's default behavior. Using inheritance, he can refine the generated proxy code in an application-specific way. The instances of such refined proxy-classes are called *smart proxies* [14]. They play an important role in Harmony, as we will see in Subsection 5.1. For the server side the IDL compiler generates so-called *skeletons*. In a nutshell, skeletons are envelopes of the server objects, where the application programmer must implement the body of the methods. In this way, CORBA hides the communication details and achieves location transparency for the client and server side.

Message passing through the ORB includes the following transformations: the ORB *marshals* all method parameters and result values into a canonical, internal representation before transmission, and *demarshals* them at the receiver. Consequently, CORBA communication is expensive, and the transport of large numbers of data items is relatively costly. The ORB marshals/demarshals for every method invocation individually. For more technical explanations, see [9]. While facilitating implementation transparency, these transformations may not only lead to a significant runtime overhead, but also to high resource consumption, as we will show in Section 5.2.

3.2. Persistent CORBA Objects

Having specified application-specific CORBA objects using an IDL schema, one frequently wants to make such objects persistent. Three techniques exist to do so.

1. The straightforward solution is an *Object-Oriented Database Adaptor (OODA)*, as defined by the CORBA standard. An OODA maps objects stored in an ODBMS into CORBA objects.

2. To provide persistence for all objects, even those not managed by a DBMS, the service architecture of CORBA includes a definition of a *Persistent Object Service (POS)*.

3. ORB-proprietary mechanisms allow for specific initializations of newly instantiated CORBA objects.

The third technique, while being ORB-specific, has the following advantages: it is a low-level ORB mechanism offering performance comparable to that of an OODA. Additionally, it provides flexibility similar to the one of a POS, e.g., it does not require an ODBMS to store the data. For the application-specific *Customer* objects in the evaluation, we have therefore pursued the third approach. Orbix instantiates CORBA objects via a generic *LoaderClass* [14]. Using inheritance, the programmer can implement a *LoaderClass* of his own for objects with persistent state. In this way, the loader can load the object's state from the persistent data source. On the other hand, this means an extra database access for newly instantiated persistent CORBA objects compared to a service object, which provides fixed access methods to certain data.

3.3. Declarative Data Access with CORBA

By means of the Object Query Service, the CORBA standard makes possible declarative access to data. In the following, we explain basic concepts and components of the OQS. It consists of seven interface definitions. In a nutshell, they can be seen as abstract object types:

The *QueryEvaluator* plays the central role. It encapsulates the query processing facilities. It publishes the query functionality through its `evaluate()` method. The specification proposes *QueryEvaluators* to support OQL or SQL dialects as query language. CORBA provides the concept of collections and associated iterators for managing bulk data. *Collection*s and *Iterator*s are again abstract object types. Please note, that the CORBA standard defines iterative access to results with synchronous invocation of the iterator methods. A naive implementation will therefore suffer from the multiple of the invocation latency explained above. In Harmony, we avoid this bottleneck, as we explain in Subsection 4.2. Further, CORBA distinguishes between implicit and explicit collections. While explicit collections store their elements explicitly, implicit collections retrieve their elements not before access time. In other words, only the iterators of implicit collections actually access the data sources. A *QueryableCollection* combines the query functionality of the *QueryEvaluator* with the data management functionality of a *Collection*. The rest of the standard interfaces of the OQS have administrative tasks.

4. The Harmony Object Query Service

In this section, we give an overview of the Harmony prototype. We start with the relevant concepts for querying in Subsection 4.1. The subsequent subsections review implementation issues and describe data source integration. Subsection 4.4 explains query evaluation with Harmony.

4.1. Harmony's Physical Query Algebra

With regard to terminology, we distinguish between query algebra operators, which are on the logical level, e.g. *join*, and operations, which correspond to implementations on the physical level, e.g., *merge join* [11]. In Harmony, a query is evaluated by a sequence of physical operations which are partially ordered. As usual, such a sequence is called the *execution plan of the query*. An execution plan has a tree structure, each node representing one physical operation. Edges connect the nodes according to their partial ordering. Each edge therefore represents the dataflow between subsequent operations. The leaves stand for the data sources needed for query evaluation. The root node represents the result of the query. Each of these physical operations produces one output stream by applying its functionality to one or more input streams[1]. Harmony evaluates an execution plan in a bottom-up fashion. There are essentially two ways to execute a plan [7]: the first one is that a node processes the whole input before the next operation starts (*set-at-a-time*). With the second approach, a node processes its input element by element, as soon as its predecessors have produced the next tuple (*tuple-at-a-time*). Tuple-at-a-time may impose problems when taking updates into account [7], but we do not consider this aspect here. We also refer to tuple-at-a-time as the *dataflow approach for query execution*.

[1]We leave aside concepts such as bypassing of results [15].

```
SELECT c.Name
FROM   File     l,
       Orders   o,
       Customer c,
WHERE  l.Word1
       = o.Key
  AND  c.CustKey
       = o.CustKey
```

$$\begin{tikzcd}
 & & \Pi_{\mathrm{Name}} & & \\
 & & \bowtie_{\mathrm{Word1=Key}} \arrow[u] & & \\
\omega_{FileSys} \arrow[urr] & & & \omega_{Oracle} \arrow[ul] & \\
\Pi_{\mathrm{Word1}} \arrow[u] & & & \Pi_{\mathrm{Name,Key}} \arrow[u] & \\
\mathrm{File} \arrow[u] & & & \bowtie \arrow[u] & \\
 & & \mathrm{Customer} \arrow[ur] & & \mathrm{Orders} \arrow[ul]
\end{tikzcd}$$

Figure 1. Example execution plan of a query over heterogeneous repositories.

In addition to the standard query operators, Harmony's physical algebra also includes *meta operators*: *wrap*, *send* and *receive*. Meta operators do not change the data stream, but perform some control function. *wrap* (ω) abstracts from the underlying storage systems. It translates the sub-tree it roots to the respective query language. E.g., the right ω_{Oracle} node in Figure 1 will translate its Π, $\bowtie$ and *extent* nodes into an SQL query to submit to the Oracle RDBMS. For more details see Subsection 4.3. The *send/receive* operators model the asynchronous set-oriented data transfer between sub-trees. The corresponding operations are always used as pairs. With regard to performance, as we show in Section 5.1, the following issues are of out-most importance: send produces its output stream in parallel to execution of operations following the receive node. This is *intra-query parallelism*, as referred to in the introduction. Furthermore, the send/receive operations do not exchange single data items, but bulks of values. This is the feature referred to previously as *bulk transfer*. The number of data items in such a bulk is subject to configuration [11].

4.2. Implementation of the Harmony OQS

Whereas in Subsection 4.1 we have looked at concepts for distributed query evaluation with Harmony, we now focus on their implementation using CORBA mechanisms. According to the OQS-specification, a client contacts the OQS with a declarative query, e.g., formulated in ODMG-OQL. In our prototype, we submit a query execution plan instead of a declarative query. The OQS returns the query result either as single value or as a CORBA collection object. For the evaluation of an execution plan, the OQS instantiates a *QueryableCollection* for each of the plan's nodes. Therefore, in Harmony we have refined the abstract *QueryableCollection* into two abstract base object types of the available query operations: *UnaryQCollection* and *BinaryQCollection*. This classification reflects the fact that each operation has either one or two input streams.

With Harmony, the dataflow approach is implemented as follows: Harmony uses only implicit collections. Intermediate results are not stored in the collections, but computed as soon as the `next()` method of an iterator requests the next result item. The only exceptions are the send/receive meta operators: the *SendQCollection* computes a chunk of result values in parallel to the consuming *ReceiveQCollection*. This is sufficient to achieve intra-plan parallelism. But we also want to exploit these concepts for data transfer from the OQS to the client. The problem is that, as default behavior, proxy objects only support synchronous method invocations. Hence, with Harmony we have come up with the following refinement: the proxy objects of iterators actually perform the receive meta operation. This is achieved through Orbix's *smart proxy* mechanism introduced in Subsection 3.1. As explained, they allow to refine the generated proxy code with proprietary code. A Harmony client temporarily caches a chunk of results in the smart proxy object of the result iterator. With each invocation of `next()`, the proxy returns one item to the client code. While the client consumes the current result chunk, the OQS already computes the next results. As we will see in Subsection 5.1, Harmony heavily benefits from this technique.

4.3. Data Source Integration

Our approach to integrate heterogeneous data repositories is to access repositories via *wrappers* [8]. They

- administer connections to a repository,
- transform OQS queries into the local query language,
- convert results into the CORBA type system, and
- export the repository's query capabilities and cost statistics (not covered in this article).

Wrappers are not part of the CORBA specification, but a logical advancement of the OQS concepts explained above. More specifically, a wrapper consists of three CORBA objects: a *QueryableCollection* implements the *wrap* operator for the underlying storage system. The corresponding *Iterator* objects carry out the actual data access. A *WrapperFactory* manages the creation of several instances of a wrapper's *QueryableCollection*. The wrapper collections apply the concept of implicit collections. When instantiating an implicit collection, only basic initializations occur. These include establishing the connection to the underlying repository. It is the iterator that carries out the actual data access, submitting the query to the database and retrieving its result.

Summing up this point, to integrate a new repository type, one has to implement just three objects. They are of types *WrapperFactory*, *QueryableCollection* and *Iterator*. The only functionality a wrapper must provide is to access data collections of its source. In the simplest case, this means the ability to scan the source. But some data sources do not support scans, e.g., WWW servers. With such sources, only limited query functionality is feasible.

4.4. Query Evaluation with Harmony

Harmony uses a peer-to-peer execution model. I.e., there is not one central object query service, but the OQS consists of several distributed objects. To maintain the system configuration, Harmony utilizes another basic CORBA service, the *Naming Service*. This service binds CORBA objects to names. All OQS objects register themselves with the Naming Service. We can dynamically divide the query evaluation between them, e.g., for load balancing. Evaluation of a query in Harmony consists of the following steps:

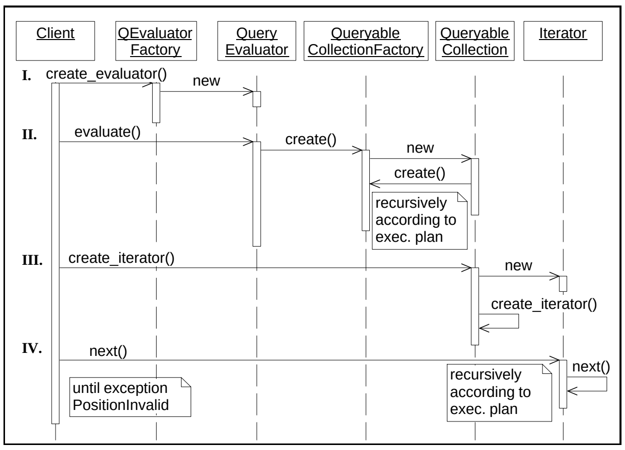

Figure 2. Interaction diagram of query evaluation in Harmony.

I. The client asks the Naming Service for the reference of the *QueryEvaluatorFactory* of the default OQS server and connects by calling the `create_evaluator()` method.

II. The client submits its query to the newly instantiated *QueryEvaluator*. The evaluator parses and optimizes the query, resulting in an execution plan. It then "instantiates" this plan, e.g., it creates for each of the nodes of the plan a *QueryableCollection* of the corresponding type. In more detail, the *QueryableCollectionFactory* instantiates a *QueryableCollection* for the root of the plan. It then invokes the `evaluate()` method of the created collection with the execution plan as argument. The collection removes its own node from this plan, parses its arguments and then recursively instantiates *QueryableCollection* for each of its input streams via the *QueryableCollectionFactory*. The advantage is that each *QueryableCollection* can be instantiated at any site in the system. This avoids a single coordinator site, through which all data is routed, e.g. for evaluating a join. Instead, even complex query operations can be executed as near as possible to the storage systems. The result of this instantiation process is a reference to the topmost *QueryableCollection*. It is returned to the client. This leads to a peer-to-peer execution model with location transparency. A quantitative assessment of this aspect of the peer-to-peer execution, notably workload of different sites, is beyond of the scope of this paper.

III. The client calls `create_iterator()` on the returned *QueryableCollection*. The collection recursively instantiates all the iterators needed and returns a reference to its own iterator.

IV. The client retrieves the query result through the iterator by subsequent `next()` calls until it encounters a *PositionInvalid* exception.

Figure 2 above illustrates these interactions between different objects in Harmony.

5. Performance Evaluation

We distinguish between three different kinds of data access: point access, context access, and bulk access. For the evaluation we use the scenario of a customer information system as a case example. The schema is part of the database schema of the TPC/D benchmark. The TPC/D database contains data of standard data types. For the moment, such data seems to be sufficient to identify the overhead of CORBA-based data transfer. In other words, we focus on the "horizontal market"-category in this paper. We compare Harmony with the two alternatives mentioned in the introduction:

- With regard to Alternative 1, we call the CORBA objects encapsulating customer data in our example the *Customer Information Service (CIS)*. This service consists of a *CIS* service object, which provides access to several persistent *Customer* objects. Its IDL is shown in Appendix A. Subsequently, we refer to this alternative as *(normal) CORBA object access*.

- Regarding Alternative 2, we use static SQL queries and the Oracle ESQL/C bulk access technique, called *array fetching*.

Harmony itself is implemented in C++ and based on the commercial CORBA implementation Orbix MT 2.3c from Iona Technologies.

Subsection 5.1 evaluates the runtime performance of Harmony and the two alternatives, both for local and remote access. Subsection 5.2 investigates the resource consumption of the approaches. Subsection 5.3 investigates access to data stored in a file and in a relational database. We have run all experiments on SUN SPARCstations under Solaris 2.5, which are connected with a 10-BaseT Ethernet LAN. The wide area network (WAN) measurements have been conducted between GMD Darmstadt, Germany, and ETH Zurich, Switzerland. All experiments have been conducted early in the morning with no network load, which was especially in favor for the WAN measurements.

5.1. Runtime Performance

In this subsection, we evaluate the runtime performance of Harmony for the point-, context- and bulk access categories. The runtimes shown include connect, data access and disconnect. We have created the partial TPC/D schema in an Oracle instance and have populated it with 150.000 customers and 1.500.000 orders. The database size, including indexes, was approximately 260 MB.

Point Data Access. The customer numbers of specific customers have been retrieved. Figure 3 shows the corresponding SQL query, CORBA object method invocations and Harmony execution plan. Figure 4(a) displays the

```
SELECT  CustKey
FROM    Customer
WHERE   Name = :n
```

```
cust =
 CIS->access(:n);
key  =
 cust->Key();
```

$$\begin{tikzcd}[row sep=small] \omega_{Oracle} \\ \Pi_{\mathrm{CustKey}} \arrow[u] \\ \sigma_{\mathrm{Name}=:n} \arrow[u] \\ \mathrm{Customer} \arrow[u] \end{tikzcd}$$

Figure 3. SQL query, CIS method calls, and Harmony's execution plan for point access.

runtime result of this experiment, where a single `integer` value has been retrieved. The overhead of the CORBA method invocation effort shows up both with Harmony and the application-specific CIS: the difference between the ESQL/C approach and the application-specific CORBA service is about 25%. Due to the complex initialization, the difference between access duration with Harmony and ESQL/C is even larger. For point access, we do not see any optimizations that could improve Harmony's runtime performance.

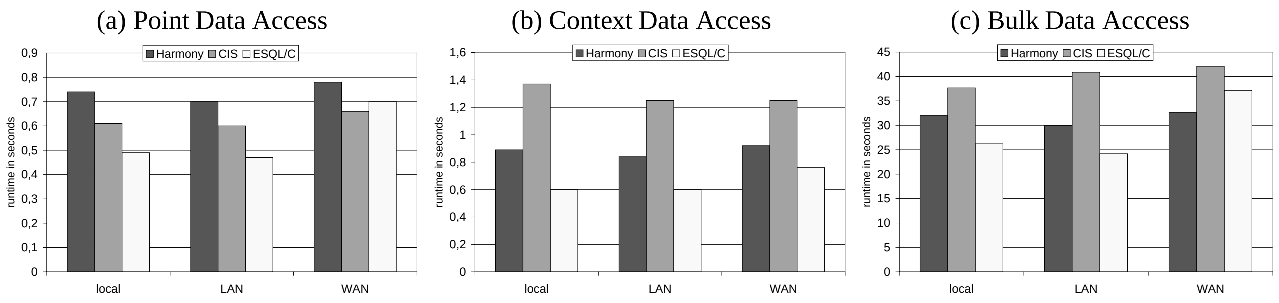

Figure 4. Local/remote data access times (seconds) for the three access categories.

Context Data Access. In the category of context data exchange we have retrieved the list of orders of specific customers, as shown in Figure 5. Both CORBA objects access and Harmony are at least 25% slower than the direct database access via ESQL/C. In contrast to the first experiment, Harmony performs better than normal CORBA objects, as the latter have to access the database twice: first the corresponding *LoaderClass* (see Subsection 3.2) initializes the customer object. Then the `orders()` method retrieves the corresponding order values from the database. This illustrates that a careless approach to IDL integration of data sources can deteriorate performance.

```
SELECT o.*
FROM   Orders o,
       Customer c
WHERE  Name = :n
  AND  c.CustKey
     = o.CustKey
```

```
cust   =
CIS->access(:n);
orders =
cust->orders();
```

$$\begin{tikzcd}
 & \omega_{Oracle} & \\
 & \Pi_{\mathrm{Key},\ldots} \arrow[u] & \\
 & \sigma_{\mathrm{Name}=:\mathrm{n}} \arrow[u] & \\
 & \bowtie \arrow[u] & \\
\mathrm{Cust.} \arrow[ur] & & \mathrm{Order} \arrow[ul]
\end{tikzcd}$$

Figure 5. SQL query, CIS calls, and Harmony's execution plan for context access.

Bulk Data Access. The corresponding experiment simulates a mass mailing activity by accessing all addresses of customers in a specific domain. Figure 4(c) presents the results. With bulk data access, the (de-)marshaling overhead is significant. The application-specific CIS is about 50% slower as a direct database access, even though here the CIS encapsulates the same database query as the one used in the ESQL/C program. On the other hand, Harmony benefits from its bulk transfer and intra-query parallelism, as introduced in Subsection 4.2. For our experiments, the Harmony client retrieves the query results in chunks of 100 values. This is the same number of items as the ESQL/C program retrieves with one of its array fetches. Without the send/receive enhancement, Harmony's execution times were around 400 seconds. Employing the send/receive intra-query parallelism, Harmony clearly outperforms the normal CORBA object access, in particular with access through a wide area network (WAN).

To summarize this part of the quantitative evaluation, both CORBA object and Harmony access times are in the same magnitude as the ones with ESQL/C. But as expected, if we compare a three-tier architecture with a two-tier approach, there is an overhead with the first alternative. The Harmony queries are about 10% to 50% slower than direct access with ESQL/C. This relative overhead is smaller than the one of accessing application-specific CORBA objects, as compared to ESQL/C, which ranges between 25% with point access and 130% with context access. This is partly due to the CORBA-inherent marshaling/demarshaling overhead and its increasing resource consumption, as we will see in Subsection 5.2. For the results with context access, another reason is our design decision to use persistent CORBA objects: first the persistent state of the *Customer* object was retrieved, then its `orders()` method accessed the database a second time. This resulted in two database accesses instead of one with Harmony or ESQL/C. Finally,

```
SELECT Address
FROM   Customer
WHERE  MSegmnt=:m
```

```
addrlist =
  CIS->addrs(:m);
```

$$\begin{tikzcd}
\omega_{Oracle} \\
\Pi_{\mathrm{Address}} \arrow[u] \\
\sigma_{\mathrm{MSegmnt}=:\mathrm{m}} \arrow[u] \\
\mathrm{Customer} \arrow[u]
\end{tikzcd}$$

Figure 6. SQL query, CIS method calls, and Harmony's execution plan for bulk access.

Harmony's query evaluation techniques are most favorable with bulk data access and through a wide area network. Here it even outperforms data access via Embedded SQL.

5.2. Resource Consumption

The experiments in Subsection 5.1 show that data exchange via CORBA is a competitive technique, if properly realized. In more detail, however, we have observed an increasing overhead with data access through application-specific CORBA objects. In another experiment we have

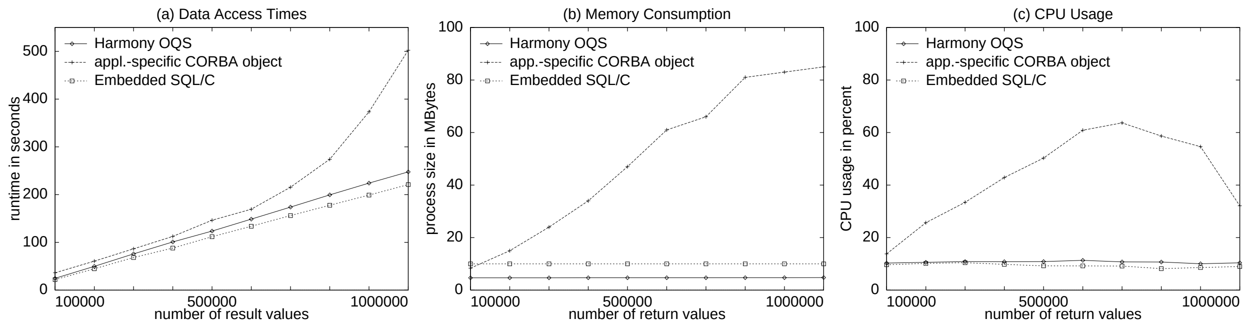

Figure 7. Access time, memory and CPU usage for increasing amounts of data.

therefore varied the amount of data being accessed up to result sets with 1 million values of 20 bytes average size. As can been seen in Figure 7(a), the runtime of application-specific CORBA objects increases more than linearly over the data size. This is because up to 20 MBytes have been transmitted, demarshaled and converted with one method invocation. As the data has to be copied for transformation at least once, this leads to an unacceptably high memory consumption both on server and client side. Figure 7(b) contains the memory consumption of the client programs, while Figure 7(c) shows the CPU usage of the clients. The increasing runtime of CORBA objects stems from their increasing memory consumption. At a certain number of return values, memory consumption is so high that the client machine is basically occupied with swapping. This is the reason for the decreasing CPU usage of the CORBA client for high data volumes, as it spends most of the time waiting for completion of disk I/O instead of computing the actual result. On the other hand, memory consumption of the Harmony client remains constant. The memory requirement of the Harmony OQS is also small in absolute terms, and smaller than the one of ESQL/C. This has two reasons: first, Harmony's data-flow execution does not produce large intermediate results. Second, the runtime library of the ORB requires less memory than the runtime environment of an Oracle client.

5.3. Heterogeneous Query Execution

The rationale behind the development of Harmony is the flexible integration of heterogeneous data sources into query execution. In this section, we assess this aspect in quantitative terms. We do not expect access times for, e.g., files to be competitive with the ones for full-fledged DBMSs. The general relevance of this series of experiments is that their results may serve as guidelines for overall system design. The case example used here is to combine 100 orders contained in a text file with the customer database. I.e., a client program identifies the customers who have submitted certain orders. Thus, in this experiment, we access two heterogeneous data sources.

The three access techniques are differently well-suited for this problem. With ESQL/C one has to write an application-specific client program. For example, the ESQL/C program parses the given text file and generates appropriate queries for Oracle. It submits a separate query for each order found in the text file, which retrieves the corresponding customer. A CIS client program looks similar. There is no suitable access method defined for the CIS to retrieve the proper customers for given orders (cf. Appendix). One has to extend the service implementation, e.g., with a method `find_customer_of(in Order order)`. The corresponding client program parses the text file and retrieves the customers using this new CIS method. Again, with this solution for each order a separate method is invoked. With Harmony, one can avoid such application-specific implementation effort. The corresponding query is shown in Figure 1. However, one needs a simple *TextfileWrapper*. This wrapper must only be capable of scanning a text file line by line. Each line is transformed into one result tuple. Harmony carries out any further processing.

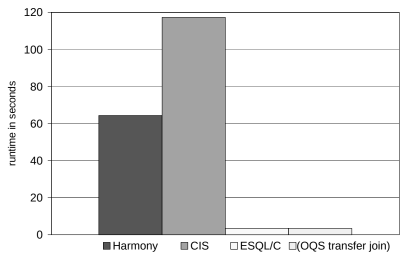

Figure 8. Runtime Efficiency for heterogeneous data access.

The runtimes shown in Figure 8 reflect our expectations. The Embedded SQL/C program is faster than the application-specific CORBA solution by order of magnitudes. Harmony's OQS is situated in between. The join method used was a *bind join* [13]. It submits separate queries to Oracle for each tuple found in the text file. To fully exploit the capabilities of Oracle, we have also implemented a new join variant, which we call *transfer join*. This join variant temporarily transfers the data from one input stream into the data source of the other. The join is then locally evaluated. In our example Oracle uses a temporary relation. This is the optimal strategy for small external data sets (here: 400 bytes), as can be seen in Figure 8: using the transfer join, Harmony's performance is comparable to the ESQL/C program in this experiment.

6. Development Complexity Evaluation

A comparison of the different access techniques is complete only when taking soft criteria, notably the implementation effort necessary as well as the lines of code needed, into account. Table 1 contains the respective numbers for the different techniques and the three access categories. We have counted the lines of code (LOC) of the clients. We have also logged the approximate programming time needed by an experienced programmer to implement the three given example queries from Figures 3, 5, and 6 using each technique. Clearly, Harmony is superior with regard

	point access	context access	bulk access
Harmony	no progtime 1 query	no progtime 1 query	no progtime 1 query
CIS	0.5 h 81 LOC	0.5 h 85 LOC	0.5 h 83 LOC
ESQL/C	0.5 h 120 LOC	1 h 168 LOC	0.5 h 144 LOC

Table 1. Implementation efforts of the three access techniques.

to these criteria. The user only has to formulate the query. Implementing a client program with Harmony requires essentially the same effort as the implementation of a client for specific CORBA objects (not shown in the table).

7. Related Work

In [5] the performance of three commercial database gateway systems – DJ, EDI/S and MIRACLE – has been studied. The evaluation concentrates on the raw SQL performance of these gateways. Neither is there a distinction between different access scenarios, nor does it take the middleware using such gateways into account, although this work is motivated by a middleware project to integrate information systems of different departments.

The evaluation of CORBA as middleware for data-intensive applications is also a topic of the PHRAMES project [20]. Similar to this paper, it classifies applications accessing data into server-oriented and data-intensive applications. It does not consider context data access and concrete measurements are not published. While Harmony's approach is more data driven, PHRAMES pursues an object-centric approach: all data items are CORBA objects. The authors focus on technical issues regarding object adaptors and the Persistent Object Service.

Similar to the Volcano prototype [12], Harmony is a query execution engine for a shared-nothing architecture. In particular, the send/receive operators have a counterpart in Volcano's *exchange* meta operator [10].

Microsoft has published a specification for a set of data access interfaces for diverse data stores based on their OLE technology [1, 18] called OLE DB. It is based on the relational data model. Queries are submitted either as SQL (the standard way) or *command trees* which vaguely correspond to Harmony's execution plans. However, the problem of dealing with different query capabilities and optimized distributed query evaluation is not addressed.

In the area of accessing and integrating distributed heterogeneous data sources, we are aware of the following recent research efforts. The Garlic system [2, 4] from IBM Almaden concentrates on the integration of multimedia information systems. The TSIMMIS project at Stanford addresses the problem of accessing non-standard data, notably semi-structured data, and proposes a flexible mediator-based approach [3, 8]. At INRIA, the Distributed Information Search Component (DISCO) has been developed [21, 22]. However, all of these prototypes focus on heterogeneous query optimization and flexible data source integration using their proprietary middleware system. In contrast to this, we have consciously decided to rely on the CORBA standard for data exchange with a focus on efficient query execution for different access categories.

8. Conclusions

We have evaluated the CORBA middleware from an information system point of view. To this end, we have built a prototypical CORBA Object Query Service called Harmony. An efficient implementation of this service is not straightforward. We have deployed classical database concepts, notably dataflow execution, bulk data transfer and intra-query parallelism.

In this paper, we have compared Harmony with other approaches to data access. We have categorized data exchange into point-, context- and bulk data access. For these three categories we have compared the performance of Harmony with the ones of application-specific CORBA objects and native Embedded SQL/C. The evaluation yields the following results: if one confines himself to CORBA's synchronous invocation model and marshaling/demarshaling overhead, execution times increase and memory requirements become unacceptable with access to bulk data through application-specific CORBA objects with single method calls. On the other hand, Harmony's implementation of the Object Query Service circumvents these bottlenecks. In our experiments, it outperforms application-specific CORBA objects for context- and bulk data access, both with respect to execution times and resource consumption. Furthermore, its runtime overhead compared to direct client/server communication via ESQL/C is low, ranging between 10% to 50%. This is remarkable because Harmony is also superior with regard to the flexibility criterion, as shown in Subsection 5.3 and Section 6.

As we have shown in this paper, efficient and flexible access to bulk data via CORBA is possible. In the future, we plan to investigate query plan generation in our environment. Furthermore, we are interested in the dependencies between transactions and query evaluation in Harmony.

Acknowledgements. We are indebted to Gustavo Alonso, Hans-Jörg Schek, and Joachim Schmidt for their valuable comments on this work. Further, we thank Dimitar Boskov, Sarah Fuchs, Rolf Neggri and Marcel Steinmann for their help with the implementation.

References

[1] J. A. Blakeley. Data access for the masses through OLE DB. In *Proceedings of the 25th ACM SIGMOD Conference on Management of Data, Montreal, Canada*, June 1996.

[2] M. J. Carey et al. Towards Heterogeneous Multimedia Information Systems : The Garlic Approach. In *Proc. 5th Int. Workshop on Research Issues in Data Engineering(RIDE): Distributed Object Management, Taipei, Taiwan*, 1995.

[3] S. Chawathe et al. The TSIMMIS project: Integrating heterogeneous information sources. In *Proceedings of the 10th Anniversary Meeting, Information Processing Society of Japan, Tokyo,*, Oct. 1994.

[4] W. F. Cody et al. Querying Multimedia Data from Multiple Repositories by Content: the Garlic Project. In *Proc. Visual Database Systems, Lausanne*, March 1995.

[5] F. de Ferreira Rezende and K. Hergula. The heterogeneity problem and middleware technology: Experiences with and performance of database gateways. In *VLDB'98, Proc. of 24th Int. Conf. on Very Large Data Bases, New York*, 1998.

[6] B. Eng. CORBAservices Feature Matrix. URL: http://www.vex.net/˜ben/corba/cosmatrix.html, Nov. 1998.

[7] G. Fischer and K. Aberer. Admissible record-oriented evaluation plans for declarative updates. In *First East-European Symposium on Advances in Databases and Information Systems - ADBIS'97, St. Petersburg, Russia*, 1997.

[8] H. García-Molina et al. Integrating and accessing heterogeneous information sources in TSIMMIS. In *AAAI Spring Symposium on Information Gathering*, 1995.

[9] A. Gokhale and D. Schmidt. Measuring and optimizing CORBA latency and scalability over high-speed networks. *http://www.cs.wustl.edu/˜schmidt/ieee_tc-97.ps.gz*, 1997.

[10] G. Graefe. Encapsulation of parallelism in the Volcano query processing system. In *Proc. of the 19th ACM SIGMOD Conf. on Management of Data, Atlantic City*, 1990.

[11] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–169, June 1993.

[12] G. Graefe. Volcano – an extensible and parallel query evaluation system. *IEEE Transactions on Knowledge and Data Engineering*, 6(1), February 1994.

[13] L. M. Haas, D. Kossmann, E. L. Wimmers, and J. Yang. Optimizing queries across diverse data sources. In *Proc. of the 23rd Int. Conf. on Very Large Data Bases, Athens*, 1997.

[14] IONA Technologies Ltd. *Orbix MT 2.3c Programmer's Guide*. Dublin, Oct. 1997.

[15] A. Kemper, G. Moerkotte, K. Peithner, and M. Steinbrunn. Bypassing joins in disjunctive queries. In *Proc. of the 21st Int. Conf. on Very Large Data Bases, Zurich*, 1995.

[16] Object Management Group. Object Management Architecture Guide. TC document 92.11.1, Sept. 1992.

[17] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, 2.0 edition, 1995.

[18] S. Rauch. Talk to any database the COM way using the OLE DB interface. *Microsoft Systems Journal*, July 1996.

[19] U. Röhm and K. Böhm. *Working Together in Harmony — An Implementation of the CORBA Object Query Service and its Evaluation*. DBS research group, ETH Zurich, 1998.

[20] J. Sellentin and B. Mitschang. Möglichkeiten und Grenzen des Einsatzes von CORBA in DB-basierten Client/Server-Anwendungssystemen. *BTW*, 1997.

[21] A. Tomasic, L. Raschid, and P. Valduriez. Scaling heterogeneous databases and the design of Disco. In *Proc. of the 16th Conf. on Distributed Computing Systems, Hongkong*, 1996.

[22] A. Tomasic et al. The distributed information search component (Disco) and the World Wide Web. In *Proc. of the ACM Int. Conf. on Management of Data, New York*, 1997.

Appendix A: CORBA IDL of the CIS

```
enum nation { Algeria, Argentina, ... };
enum market { Automobile, Building, ... };
struct Order{long   OrderId;string Date;
             float  Price;  string Priority;
             char   Status; string Clerk;
             string Comment; };
interface Customer
{
  readonly attribute long   Key;
  readonly attribute string Name;
  readonly attribute string Address;
  readonly attribute nation Nation;
  readonly attribute string Phone;
  readonly attribute float  AccountBalance;
  readonly attribute market MSegmnt;
  readonly attribute string Comment;
  sequence<Order> orders();
};
interface CIS
{
  Customer         access(in string name);
  sequence<string> addrs (in string mseg);
};
```