A Case for Efficient Portable Serialization in CORBA
(Preliminary Version)

Roy Friedman∗ Erez Hadad
Computer Science Department
Technion - Israel Institute of Technology
Haifa 32000
Israel
E-mail: {roy,erezh}@cs.technion.ac.il

Abstract

This technical note reports on the dramatic improvement in performance that was obtained in FTS, a CORBA replication service, by simply replacing the standard portable serialization and de-serialization mechanism with an optimized one that is still portable. The surprising result is that although serialization and de-serialization only accounts for a marginal fraction of the latency in serving clients’ requests, both the throughput and the overall latency gains were dramatic. This paper analyzes the default mechanism vs. the optimized one, and explains this counter-intuitive result. Interestingly, the lessons from this work are valid for any multi-threaded server based system.

1 Background

Providing highly available fault-tolerant services using client/server middleware standards involves two main tasks: The first is to ensure that client requests always reach an alive server while hiding the low level details of this from the client’s application code. In particular, for scalability and portability reasons, it is desired that client invocations will reach one of the servers using the default RPC-style communication mechanism of the middleware standard being used. The second issue is how to keep the service’s state consistent across all replicating servers.

In this paper we focus on FTS [2, 4], which is a CORBA [10] active replication service. FTS is based on the following design choices: Servers publish their references, or IORs to be exact, on a public medium such as a naming service or a web site. When a client wishes to communicate to a replicated service, it obtains the IOR from the naming service in the standard CORBA manner. Following this binding process, the client invokes remote methods using the standard IIOP protocol as implemented by the ORB. However, if the ORB raises an exception that indicates that the server might have crashed, or that the underlying TCP connection was disrupted, a portable interceptor [13] registered at the client’s ORB catches the exception and redirects the request to another server. The IOR of the alternate server is obtained through a fault-tolerant redirection sub-service of FTS called LMR [5]. More precisely, since CORBA’s portable interceptors standard does not allow an interceptor to reissue a request by itself, the interceptor raises a redirection exception. Such exceptions are caught by the ORB before returning control to the application, and then the ORB automatically reissues the request to the new server [3]. This is done completely transparently to the client application’s code.

As for servers consistency, FTS overwrites the default Portable Object Adaptor (POA) with its own Group Object Adaptor (GOA) [4]. Internally, GOA utilizes the standard POA, while externally it also exports the same interface as the POA. When a client’s method invocation is received at the server, FTS’ GOA relays the request to the other replicas through a totally-ordered reliable multicast channel provided by an underlying group communication (GC) component, e.g., Ensemble [6]. The reason for doing this above the ORB level, rather than inside a server side portable interceptor, is that CORBA prevents a portable interceptor from having access to the request data in security aware languages like Java. The only way for a standard portable code on the server

∗Partially supported by an IBM faculty award

1If a request is tagged as a non-update, then the request is served locally.
side to see the request’s data is by passing it through a DSI [10] “proxy” object, which is what the GOA does internally. Furthermore, the request data is only available at its deserialized form, so in order to send it through the GC subsystem, we need to serialize it again. In order to remain platform-independent and portable, FTS utilizes CORBA’s portable serialization and de-serialization. Basically, the process involves dynamically intercepting the request’s data using the DSI servant, then serializing the request data and passing it to the GC subsystem. Upon reception, update requests are de-serialized from the buffers delivered by the GC. Each request data is then used to construct a dynamic invocation (DII) [10] that is locally invoked upon the matching object replica.

As becomes evident from the above discussion, FTS operation relies heavily upon CORBA portable serialization and de-serialization, which appeared to be somewhat inefficient. Thus, we examined how can this aspect be improved. This paper reports on our findings. In particular, we have discovered that while serialization and de-serialization only take a minor fraction of the overall latency in servicing clients’ requests, our optimization resulted in a dramatic improvement in both throughput and latency. In order to understand this surprising result, we analyze the latency factors in handling a request, and characterize them as either CPU, I/O, or waiting. We then use the above analysis to explain that optimizing CPU intensive activities can result in the behavior we observed, in particular since serialization and de-serialization are indeed such activities.

The rest of the report is organized as following: Section 2 describes the standard portable serialization and de-serialization methods offered by CORBA. Section 3 focuses on the implementation of those standard methods in ORBacus [1], our test ORB. Section 4 presents our optimized yet portable method for (de)serialization. Section 5 describes a comparative evaluation of all the methods in FTS with respect to throughput. Last, Section 6 concludes this paper.

2 CORBA Portable Serialization

Generally, CORBA provides serialization and de-serialization of CORBA IDL entities to and from byte-sequences. Thus, in order to serialize an object’s state through CORBA, the state must be defined in IDL. Two methods are available for this: IDL struct (data-only record) and IDL valuetype, which is a full object with methods that is re-created by a user-defined factory upon de-serialization. Our specific need was to serialize a CORBA dynamic request, which is not a pure IDL entity, but rather a “pseudo-object” bound to the local implementation. Therefore, serializing a request using an IDL struct requires a middle-stage of reading and writing the request object data to and from a CORBA struct. Also, we chose not to declare a pure CORBA valuetype encapsulating the dynamic request data, since some of the request’s most crucial data, such as its parameters, are also stored in non-serializable CORBA NVList pseudo-objects when retrieved from the DSI-based proxy object. Thus, valuetype serialization would require another middle-stage of conversion to a serializable format. Instead, we used a more efficient alternative of declaring a CORBA custom valuetype, with customized serialization and de-serialization, thus allowing direct transfer of data from the pseudo-objects to the CORBA low-level serialization/de-serialization streams. We implemented both the struct-based and custom valuetype-based methods.

From the programming perspective, portable serialization in CORBA takes two steps. First, the CORBA entity is copied into an instance of a CORBA Any, which is a generic container for CORBA data. This step includes the actual serialization inside the Any. The second step is to encode the Any’s contents into a byte-sequence using a CORBA Codec. The second step is required because the CORBA standard supports both little- and big-endian byte ordering of serialized values, so the serialized byte stream must be prefixed with a special tag byte indicating which of ordering variants is used. This is called CDR encoding. Portable de-serialization takes similar two steps in reverse order.

It should be noted that specifically for CORBA/Java mapping, a “CORBA portable OutputStream” [12] is also available. This OutputStream is a stream-like interface to low-level non-CDR serialization. After writing some CORBA data to it, it can create a “CORBA portable InputStream” from which data can be read, either as the original CORBA types or as a byte sequence (serialized form). The creation of the InputStream involves a copy of the OutputStream contents to the InputStream because the standard demands that both instances become independent after creation. The CORBA standard recommends the use of these streams mainly for internal ORB operation, despite them being standard and publicly available to users. As this report will show, utilizing these unknown streams was the key to improving FTS throughput while keeping it portable.
3 Serialization in ORBacus

Next, we carefully reviewed the CORBA serialization implementation inside the ORB we use for testing, which is ORBacus [1] for Java v4.1.3. For the purpose of the following analysis, we define a round as a single pass over the entire data that undergoes (de)serialization for the purpose of copying/encoding/decoding into another buffer. According to the previous section, efficient portable (de)serialization should take no more than two rounds, one for the Any and one for the Codec.

Serialization of structs (and also unions, arrays and a few other aggregates) seems to be quite inefficient in ORBacus. Insertion into an Any costs four rounds, which results from one round of marshalling (actual serialization) and four more rounds of copying the CORBA portable OutputStream containing the marshalled data into an InputStream and vice-versa twice, ending with a second-copy InputStream stored inside the Any. The CDR encoding operation adds two more rounds: the first copies the Any’s InputStream into an OutputStream prefixed with a CDR tag, and the second copies the byte buffer underlying the OutputStream into a new buffer that is returned to the user level. This totals to six rounds, at least four of which seem redundant considering that the internal ORB operation can access the internal implementation of every element, including the Input/Output streams.

De-serialization of structs takes five rounds, two of which in the CDR-decoding, consisting of copying an InputStream created from the byte buffer into another through a middle OutputStream. The other three rounds are in the extraction of the struct from the Any, where the internal InputStream is again copied to another InputStream through a middle OutputStream, and then unmarshalled.

In contrast, the valuetype-based serialization seems quite efficient: inserting into Any takes no rounds since Any simply stores a reference to the object to be serialized. The encoding step takes two rounds, one of which is the actual marshalling and adding of the CDR tag, and the other is copying the underlying buffer to a second one to be returned to the client. De-serialization takes only one round: during the decoding, the new object is constructed directly from an InputStream created from the incoming byte buffer, and the reference to the new object is placed inside the Any. Upon extraction, Any simply returns the inner reference. In summary, this makes the serialization cost a total of two rounds while de-serialization costs only one round.

However, the round-based analysis is insufficient to compare valuetype serialization to other methods, as is later also proven empirically. This is because the algorithms in use involve more than a linear processing of data. Generally, a valuetype can contain other valuetypes with possible references between themselves. As a result, serializing a valuetype is a graph linearization problem.

The standard defines a complex serialization algorithm that features compact output (detecting and suppressing duplications and rings). Furthermore, the algorithm allows splitting the serialized data into “chunks” to facilitate incremental processing both at the sender and at the receiver. Consequently, the algorithm incurs much more processing and memory overhead for caching and scanning purposes, both during serialization and de-serialization.

Important Comment: We would like to emphasize that the above discussion is only valid w.r.t. portable serialization carried above the ORB. Internally, ORBacus has a very efficient one pass serialization scheme, which is applied when messages are sent and received from the network.

4 Optimizing Portable CORBA Serialization

After having estimated the costs of using standard CORBA portable serialization with ORBacus, we realized that we could significantly reduce these costs, at least compared to the struct-based method, without losing portability. This can be done by directly invoking the CORBA portable OutputStream and InputStream. For serialization, we marshal the fields of the request data directly to an OutputStream prefixed with a CDR tag byte. We then create an InputStream from it, read it as a sequence of bytes and copy it to a byte buffer. This is achieved with a total of three rounds. De-serialization is analogous but in reverse order and costs three rounds as well. Based on our earlier analysis, this method requires roughly 50% and 40% less processing for serialization and de-serialization, respectively, compared to the struct-based method.

The optimized method is still portable across different ORBs since it uses standard CORBA/Java provisions. Furthermore, it can even be ported to other language mappings where portable streams are unavailable, since the implementation of the streams is also portable. This is because the marshalling and CDR encoding techniques are defined as part of the core standard. Constructing a custom but portable serialization/de-serialization mechanism

---

2 Copying an InputStream into an OutputStream is done through an internal ORBacus method.
would also allow for further optimizing the method. The resulting mechanism could perform serialization and de-serialization in a single round. It may even be considered portable to invoke vendor-proprietary ORB code that implements the standard’s algorithms.

The main drawback of this method is that unlike standard portable serialization, it is not generic, due to lack of support from an automatic code-generation mechanism such as an IDL compiler. Thus, performing serialization and de-serialization using our proposed method requires manual construction of the marshalling and un-marshalling code.

5 Performance Evaluation

5.1 Environment

Hardware: The hardware platform on which FTS was tested is an IBM JS20 Blade Center with 28 blades. Each blade has two PowerPC 970 processors running at 2.2GHz with 4GByte RAM. The blades are divided into two 14-node pools, each internally connected with a Gbit Ethernet switch, and both switches are connected to each other. Additionally, the cluster includes an Intel X346 server with dual Pentium 4 Xeon 3.0GHz (hyper-threaded) with 2GByte RAM, used also as an NFS server for the file system on which FTS is installed.

Software: The blades and the server run Suse Linux Enterprise Server 9 (2.6.5 kernel). All FTS components run on IBM JVM v1.4.2 SP2 for PPC and Intel platforms, respectively. The ORB used is ORBacus for Java v4.1.3. Ensemble v2.01 is used for group communication.

FTS Deployment: Clients are running in blade pool 1, servers in blade pool 2. Each client blade machine runs 250 clients sharing a common ORB. Each server blade machine runs a single server and a local Ensemble daemon. A single LMR is installed on the Intel server for the purpose of constructing the server cluster. Also, a single ORBacus interface repository daemon runs on the Intel server. Last, the Intel server runs a test manager, whose purpose is to conduct a single test of clients vs. servers and an executive that generates sequences of tests and collects the results.

5.2 Procedure

The first test was a simple linear test of performing serialization and de-serialization using each of the methods. Each (de)serialization operation was repeated 10000 times, 10 times over. The results are in Table 1. Note that Table 1 also contains an estimate of the average time required to process a request in an FTS cluster with \( n \) servers, in the form of \( D + S/n \). This is because each request is serialized once only in one server, but de-serialized once in each server. The request processed is an update operation with a single string parameter of 15 characters. The estimate was computed for clusters of 2 and 4 servers so as to match the second test.

The second test was a throughput measurement in a highly multithreaded environment under high load. Each server uses a 500-thread pool to handle client requests. The test cases were run with 1000 and 2000 clients vs. clusters of 2 and 4 servers, all invoking the same request as in the first test. For each test case, a series of 3 runs is conducted. Each run begins with a warmup phase during which each client performs 500 invocations, then continues with a “tail” of additional batches of 100 invocations until all clients are past warmup. This is when the servers are assumed to be at peak load and the test begins. During the test each client performs another sequence of 500 invocations and then a “tail” to maintain server load until the last client is past the test, after which all the clients and servers shut-down. Throughput is measured by having all the clients measure their average request round-trip time during the test phase. The average of all the clients is averaged again over all the runs. The results are in Table 2. Additionally, for each method and test case, the ratio of \( \frac{D+S/n}{RTT} \) is computed as an estimate of how
Table 2: Throughput test results (req/sec)

<table>
<thead>
<tr>
<th>Test Case</th>
<th>Valuetype</th>
<th>Struct</th>
<th>Optimized</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Throughput</td>
<td>Throughput</td>
<td>Throughput</td>
</tr>
<tr>
<td></td>
<td>Ratio</td>
<td>Ratio</td>
<td>Ratio</td>
</tr>
<tr>
<td>1000/2</td>
<td>1217.83</td>
<td>1174.81</td>
<td>1843.25</td>
</tr>
<tr>
<td>1000/4</td>
<td>470.76</td>
<td>507.11</td>
<td>1430.15</td>
</tr>
<tr>
<td>2000/2</td>
<td>970.63</td>
<td>948.88</td>
<td>1771.61</td>
</tr>
<tr>
<td>2000/4</td>
<td>750.89</td>
<td>655.34</td>
<td>1100.16</td>
</tr>
</tbody>
</table>

Table 3: Summary of improvements with optimized (de)serialization

<table>
<thead>
<tr>
<th>Method</th>
<th>Cluster Size</th>
<th>Time Reduction $\Delta T$</th>
<th>Expected Throughput Gain $\frac{1}{\Delta T}$</th>
<th>Measured Throughput Gain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Valuetype</td>
<td>2</td>
<td>16.8%</td>
<td>120.19%</td>
<td>151.35%</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>21.6%</td>
<td>127.55%</td>
<td>303%</td>
</tr>
<tr>
<td>Struct</td>
<td>2</td>
<td>27.5%</td>
<td>138%</td>
<td>156.89%</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>27.33%</td>
<td>137.6%</td>
<td>282%</td>
</tr>
</tbody>
</table>

much time does the serialization plus de-serialization work takes out of the entire request processing path through the server.

5.3 Analyzing The Results

5.3.1 The Gains in (De)Serialization Compute Time

Table 1 and Figure 1 exhibit a comparison between the serialization methods and also tests the consistency between our analysis of the (de)serialization methods and the actual implementation. The optimized serialization is in practice only 28.4% cheaper than the struct-based method and the optimized de-serialization is only 27% cheaper compared to the struct-based de-serialization, as opposed to 50% and 40% reduction based only on the round comparison. This points some crudeness of our analysis as it ignores additional time costs such as those of dynamic memory allocation for buffers and streams during both serialization and de-serialization.

The valuetype-based de-serialization proves to be about as efficient as the struct-based method. However, the valuetype-based serialization proves to be much better than the struct-based approach, and even slightly better than the optimized serialization. As mentioned before, round-based analysis is insufficient to properly evaluate this method, as is clearly demonstrated in the de-serialization cost which is much higher than the mere cost of a single round. However, some justification for the efficient serialization comes from that for simple small non-recursive valuetypes, the serialization cost seems close to two rounds, ignoring the memory allocation costs.

On a general note, Table 1 proves our optimized method to be considerably more efficient than the standard methods. Specifically for FTS, where each request is serialized only once but de-serialized in each server, the difference is quite significant, as can be seen by comparing the total work estimates in the last two columns: for a cluster of two nodes, the optimized method provides a 16.8% and 27.5% time reduction compared to the valuetype and struct methods respectively. For a cluster of four nodes, the respective improvements are 21.6% and 27.33%. For larger clusters, the total work improvement in FTS is expected to approximate the de-serialization reduction ratio as the serialization impact becomes negligible.

5.3.2 Throughput Gains

Table 2 shows the positive impact of optimizing the (de)serialization operations for clusters of sizes 2 and 4, and for client load of 1,000 and 2,000 clients. The table also lists the corresponding ratios of the request serialization + de-serialization out of the average total request processing time in a server. The latter highlights the fact that serialization and de-serialization take only a minor fraction of the latency in serving clients’ requests.

As can be seen, the throughput gains are very substantial. Moreover, the throughput gain increases both when the cluster size increases and when the client load increases. Interestingly, the proportional cost of serialization and de-serialization decreases in these cases, and their relative cost is in fact higher in the optimized case. These results may seem puzzling, perhaps even self-contradictory. At least we thought so at the beginning. But after
going over our code and running repeated tests that ended up with similar results, we came to the conclusion that there has to be a rational explanation for this phenomenon. But what can it be?

5.3.3 A Simple Attempt to Explain the Results

Our initial attempt to explain our results was to notice that the tasks involved in executing a request can be largely classified as either CPU intensive tasks and I/O intensive tasks, as illustrated in Figure 3. In this figure, \( S_1 \) denotes the cost of serialization and de-serialization in the standard implementation, \( S_2 \) denotes the cost of serialization plus de-serialization in our optimized implementation, and \( I \) denotes the rest of the latency. Clearly, due to multi-threading, when one request is waiting for I/O, other requests can run their CPU intensive parts using other threads. Thus, the maximal obtainable throughput of the system with an unlimited number of threads should increase like the ratio between \( S_1 \) and \( S_2 \) (rather than, for example, the ratio between \( S_1 + I \) and \( S_2 + I \)). Indeed, in our system, serialization and de-serialization are one of the major CPU-bound activities. So, does this explain our results?

Table 3 and Figure 2 exhibit the expected improvement under the above simple execution model vs. the actual improvement we measured. The problem is that the improvements are much larger than can be expected from the above model, even if we assume that the only CPU bound activity is serialization and de-serialization. So we need a better explanation.

5.3.4 A More Accurate Execution Model

In fact, our experiments did not try to measure the maximal attained throughput of the system, as doing this would have required more clients resources than we have. Instead, we used a fixed number of clients threads, each invoking its requests sequentially. Thus, the throughput that is reported here is inversely proportional to the average request service time experienced by the clients. In other words, the improved throughput that we measured also means that the overall service time was reduced by large factors. And this is as a result of a slight improvement in a CPU intensive activity that accounts for less than a permille of the overall latency. So how can this be?

A closer look at the execution model of an application in a multi-threaded environment identifies three types of activities (See Figure 4): CPU bound activities, denoted by \( C \); I/O operations, denoted by \( I \); and the time \( W \) spent waiting for the CPU when it is busy with other threads. Assume we can model the service time at the CPU and the arrival of new requests as Poisson processes. Then, from queuing theory, it is known that \( W \) grows to
infinity as the inter-arrival rate approaches the service rate (see Figure 5). This actually happens during a high load; in such situations, any small change in \( C \) implies a huge difference for \( W \).

While we have not shown that indeed the arrival rates of requests to the CPU and the service time at the CPU behave like Poisson processes, this model explains well the phenomena we observed. In particular, when the client load increases, the inter-arrival rate grows. In these cases, both the obtained improvement becomes higher and the relative cost of serialization and de-serialization becomes lower.

5.3.5 More Comments

It is important to note that the design of FTS has also contributed to the scale of the above phenomena. In FTS only a single thread is allowed to perform de-serialization of requests, in order to maintain inter-server consistency at all FTS consistency modes. That thread then passes the requests to a multi-threaded dispatcher that utilizes both processors of the test platform\(^3\). Thus, improving de-serialization has an effect of increasing both processors' utilization. Another contributing factor is the reduced locking during the serialization operation\(^4\).

An interesting aspect of our measurement methodology is that as the service time becomes smaller, the inter-arrival rate becomes higher. However, the increase in the inter-arrival rate is much smaller, since the overall remote method invocation time includes the network latency, which is not affected by our changes. We believe that this behavior is also true in many real servers. Intuitively, the faster the server serves a file, or an HTTP document, the faster the client is likely to submit his next request. Also, fast servers earn good reputation, which attracts more traffic to them.

6 Conclusions

This paper presented an alternative method for performing portable object serialization and de-serialization in CORBA and measured its impact on FTS, an active object replication service. The new method proved to be more efficient than existing methods both analytically and empirically. Employing the new method caused FTS throughput to increase dramatically, despite the small part that serialization/de-serialization take out of the entire request processing cost. The throughput increase is attributed to improved processor utilization in a highly multi-threaded environment, as well as shorter waiting queues in a multi-processor environment. This proves again the importance of identifying throughput-related bottlenecks not as the largest latencies measured along an execution path, but rather as the execution parts that are CPU-bound.

One major drawback of the new method is its lack of automatic support for varying object types. This is not much of a problem in FTS where this method is applied to a single fixed type of object, namely the dynamic request. However, a system that capitalizes on CORBA serialization for mobilizing data of various types would considerably benefit from a mechanism that could automatically analyze a data structure and generate a matching

---

\(^3\)The dispatcher used for the test is an object-sequential dispatcher that allows parallel execution of requests targeted at different objects.

\(^4\)Mutual-exclusion during serialization is used for batching multiple requests together to the same batch message.
efficient serialization/de-serialization code, such as an IDL compiler or meta-programming/reflection-based techniques such as those suggested in [7, 8]. To a larger extent, the CORBA standard itself would benefit from adding a set of guidelines/primitives for direct and efficient serialization and de-serialization of CORBA entities to and from byte-sequences. Efficient serialization and de-serialization are important not only for realizing object replication, e.g., as required by FT-CORBA [9], but also for many other applications, such as CORBA’s Externalization Service [11] or Persistent State Service (PSS) [14].

References