

Towards a Performance Model for Special Purpose ORB Middleware *

Venkita Subramonian, Guoliang Xing, Christopher Gill, and Ron Cytron
{venkita,xing,cdgill,cytron}@cse.wustl.edu
Department of Computer Science and Engineering
Washington University, St.Louis,MO

Abstract

General purpose middleware has been shown effective in meeting diverse functional requirements for a wide range of distributed systems. Advanced middleware projects have also supported single quality-of-service dimensions such as real-time, fault tolerance, or small memory footprint. However, there is limited experience supporting multiple quality-of-service dimensions in middleware to meet the needs of special purpose applications. Even though general purpose middleware can cover an entire spectrum of functionality by supporting the union of all features required by each application, this approach breaks down for distributed real-time and embedded systems. For example, the breadth of features supported may interfere with small memory footprint requirements.

In this paper, we describe experiments comparing application-level and mechanism-level real-time performance of a representative sensor-network application running on three middleware alternatives: (1) a real-time object request broker (ORB) for small-footprint networked embedded sensor nodes, that we have named nORB, (2) TAO, a robust and widely-used general-purpose Real-Time CORBA ORB, and (3) ACE, the low-level middleware framework upon which both nORB and TAO are based. This paper makes two main contributions to the state of the art in customized middleware for distributed real-time and embedded applications. First, we present mechanism-level timing measurements for each of the alternative middleware layers and compare them to the observed performance of the sensor-network application. Second, we provide a preliminary performance model for the observed application timing behavior based on the mechanism-level measurements in each case, and suggest further potential performance optimizations that we plan to study as future work.

Keywords: Real-Time Middleware, Distributed Embedded Systems, Sensor-Actuator Networks.

*This work was supported in part by the DARPA NEST (contract F33615-01-C-1898) and PCES (contract F33615-00-C-1697) programs. This paper is submitted to IEEE for purposes of journal review only, and is under review by Boeing prior to final publication.

1 Introduction

General purpose middleware based on standards such as CORBA, EJB, COM, and Java RMI now caters to the requirements of a broad range of distributed applications such as hot steel rolling mills, internet commerce, and military command and control [1]. Different kinds of general purpose middleware have thus become key enabling technologies for a variety of distributed applications.

To meet the needs of diverse applications, general purpose middleware has tended to support a *breadth* of features, with several *layers* of middleware commonly seen in large-scale applications [1]. However, simply adding features and layers is ill-suited for certain kinds of applications. As we have noted in previous work, features are rarely innocuous in applications with requirements for real-time performance or small memory footprint [2]. Instead, every feature of an application is likely to either contribute to or detract from the application in those dimensions, and it is thus crucial to study the advantages and disadvantages of each feature carefully.

For sensor-network applications, the focus of our recent research on special-purpose middleware, there is a fundamental tension between middleware solutions that are (1) general to increase portability and reduce programming cost and error rates, and (2) customized to provide stringent quality-of-service assurances. TAO [3] seeks to strike a balance between these two, with performance optimization and real-time assurance mechanisms provided as first-class features. Similarly, nORB seeks to achieve real-time performance that is similar to TAO, while reducing footprint significantly [2]. While it is clearly possible to use ACE directly, our experiences developing the versions of our example sensor-network application for ACE, TAO, and nORB, and then conducting the experiments reported in this paper led to two key observations:

- Implementation was significantly more complex using ACE instead of TAO or nORB.
- The version with ACE was initially *less* efficient than with TAO or nORB due to application of a sub-optimal concurrency design pattern.

Interestingly, the solution to the second problem was to use TAO as an exemplar for which patterns to use, which in turn resulted in the superior performance of the solution using ACE reported in this paper.

The key benefits of developing applications based on robust and pattern-rich ORB middleware are thus twofold. First, applications are simplified by abstraction and encapsulation of low-level concurrency and communication details within the middleware. Second, performance limitations resulting from potential mis-application of strategic design patterns can be avoided through use of middleware that correctly implements and applies necessary patterns. Having explained why ORB middleware is beneficial for distributed applications, we now turn our attention to the primary focus of this paper: quantitative performance comparison and preliminary performance modeling of a representative sensor-network application implemented on ACE, TAO, and nORB.

This paper is structured as follows. Section 2 describes our example sensor-network application, implemented as a distributed graph coloring algorithm using asynchronous message passing between concurrent processes. Section 3 describes the design of our experiments to quantify application and middleware performance in a realistic setting. Section 4 presents the results of our experiments, and offers a preliminary performance model based on analysis of those results. Section 5 describes related work on special-purpose middleware. Finally, Section 6 offers concluding remarks and describes future work motivated by the results presented here.

2 Special Purpose Applications

Systems of distributed networked sensors are being used in a variety of different applications ranging from temperature monitoring to battlefield strategy planning [4]. Systems in this domain are characterized by the following properties: (1) highly connected networks of (2) numerous memory-constrained endsystems, with (3) stringent timeliness requirements, and (4) support for adaptive reconfiguration of computation and communication elements and their associated timeliness requirements. Sensor networks thus challenge classical approaches to distributed computing and represent an active research area with many open questions.

This section introduces a real-world sensor-networks problem and distributed algorithmic approaches that have been applied to solving that problem. It then describes the resulting sensor-networks application we used in the middleware performance experiments described in Section 3. Section 2.1 describes a real-world problem for power-constrained sensor networks called ping node scheduling, in which a suitable schedule of node communication (similarly, for node on and off cycles) is determined. Section 2.2 gives an overview of distributed constraint satisfaction algorithms, and describes how the ping node scheduling problem can be solved using a distributed graph coloring algorithm. Finally, Section 2.3 describes the DBA-color application used to implement distributed graph coloring for ping node scheduling, including its stable sequence of computation and message passing steps that is the basis of our experiments described in Section 3.

2.1 Ping Node Scheduling

Sensor network applications, *e.g.*, for vibration damping [2], often need to schedule the limited computation and communication resources in the network. For example, to identify the current vibration mode of the structure, a *System Identification* component in the vibration damping application would send *ping* data to sensor nodes located on the structure to be damped, and would identify the vibration mode based on the response data from the sensor nodes. Since sensors and actuators run on limited energy resources, even in wired sensor networks as described in [2], the number of responding nodes, called *ping nodes*, should be as small as possible and still cover the overall area to be monitored [5]. Moreover, the signaling actions of two overlapping ping nodes should be synchronized so that no interfering signals will be generated. The problem of finding a schedule for ping node responses can be solved by constraint satisfaction techniques [6]. In this paper, we use the problem of scheduling the pinging activities of sensor network nodes to compare the performance of our special purpose middleware to that of the general purpose TAO ORB.

2.2 Distributed Constraint Satisfaction

A Constraint Satisfaction Problem (CSP) [7] aims to find consistent assignments of values to a set of variables,

whose inter-dependencies represent the constraints of a problem. For scalability reasons, distributed algorithms are more effective than centralized ones in large sensor networks, and it is thus desirable to apply a distributed approach to constraint satisfaction problems such as ping node scheduling. In a distributed CSP, variables and constraints are distributed among multiple nodes [8]. Distributed algorithms like the *distributed breakout* [8] algorithm (DBA) and its variations [6] have been shown to be very effective for solving distributed constraint satisfaction problems in sensor networks [6].

In particular, the ping node scheduling problem can be formulated in terms of a well-known distributed CSP: *distributed graph coloring* [6]. In *distributed graph coloring*, the goal is to find a valid color assignment for all vertices of a graph, each an autonomous node in a distributed network, and the constraint being that two adjacent vertices (*i.e.*, two vertices connected by an link) cannot be assigned the same color. In the context of the ping node scheduling problem, the network of sensors corresponds to a graph, a sensor-actuator node corresponds to a vertex in the graph, and connections between sensor-actuator nodes are represented by edges in the graph. The time slot scheduled for a ping node corresponds to a vertex color assignment in the distributed graph coloring problem. The example application used in our experiments described in Section 3 applies a DBA [8] to solve the *distributed graph coloring* problem, and is thus representative of sensor network CSP applications more generally. We use the term *DBA-color* for the algorithm used by our test application, as described in Section 3.

2.3 DBA-color Application

Each Node represents a distributed vertex of the graph. The sequence of events is as follows:

1. A NodeRegistry loads the graph from a file.
2. Nodes register with the NodeRegistry.
3. NodeRegistry returns neighbor data to each Node.
4. Nodes run DBA-color until a termination condition.

A group of 25 Node processes was executed on each of the 4 machines and the NodeRegistry was executed on one of the machines. A Node communicates with its neighbors by sending *parameter messages*. There are two

types of *parameter messages*: (1) *value* messages, containing the current color assignment of the sending node, and (2) *improvement* messages, containing the maximal reduction in conflicts that could be achieved by a color change at the sending Node.

Initially, every Node picks a random color from a color set of size equal to the diameter of the graph. For example, the diameter of the 100-node mesh described in Section 3.2 is 18. Each Node first sends its current color to its neighbors. If two vertices connected by an edge have the same color, then the constraint represented by the edge is considered to be violated. After receiving individual colors from all its neighbors, each node computes the extent of such violations locally and tries to minimize violations by searching for a different *candidate* color assignment. It then sends an *improvement*, which is a measure of its maximum possible reduction in violations, to its neighbors. After receiving improvements from all its neighbors, a Node will only change its color to its *candidate* color if its own locally computed improvement is the maximum among all its neighbor Nodes. This sequence of steps, called a *cycle*, is repeated until all violations are eliminated, *i.e.*, a valid color assignment is found for every Node. At this point, the algorithm is said to have *converged* and all Nodes terminate and output their final colors. Figure 1 in Section 3.1 illustrates the message interactions between a node and one of its neighbors in one cycle of the DBA-color algorithm. The original DBA algorithm is explained in detail in [8].

3 Experimental Evaluation

In this section we describe a set of experiments conducted to quantify fine-grain middleware performance, and use those results to construct a realistic model of observed behavior of the DBA-color application over different topological configurations. Section 3.1 first gives a detailed description of the performance segments of interest in the DBA-color application. Section 3.2 then describes the experimental platform used to conduct the experiments presented in this paper. Section 3.3 describes the comparison metrics used to design the experiments and evaluate our empirical results. Finally, Section 3.4 gives details of our experimental methodology that help ensure our observations are consistent, reproducible, and relevant.

3.1 Application Segments

Figure 1 illustrates the essential segments on each of two connected nodes in the DBA-color algorithm, and shows the messages exchanged between them. Each node per-

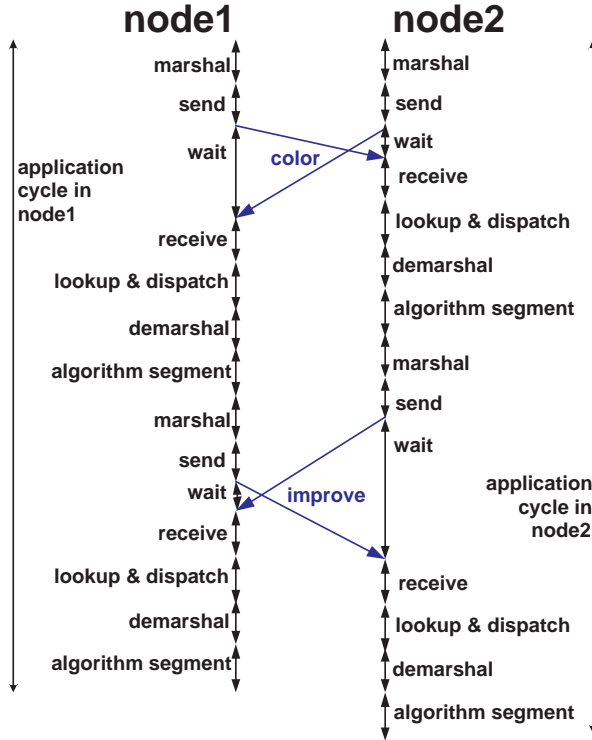


Figure 1: DBA-color Timing Sequence

forms the following sequence of steps in each DBA-color cycle:

1. it marshals its *color* value,
2. sends a color value message to each neighbor,
3. waits for color value messages from its neighbors,
4. receives each neighbor's color value message,
5. looks up and dispatches each message's method,
6. demarshals the color value from each message,
7. (algorithm segment) decides its best improve value,
8. marshals its new *improve* value,
9. sends an improve value message to each neighbor,
10. waits for improve value messages from its neighbors,

11. receives each neighbor's improve value message,
12. looks up and dispatches each message's method,
13. demarshals the color value from each message, and
14. (algorithm segment) decides its new color value.

Figure 1 highlights the fact that while some steps are synchronous within a process, *i.e.*, steps 1, 5-8, and 12-14, other steps are asynchronous, *i.e.*, steps 2-4, 9-11. Furthermore, as Figure 1 illustrates this asynchrony can lead to variations in cycle times, both between and within nodes.

Steps 3 and 10 are asynchronous due to network transmission variability: this holds generally for distributed systems with decoupled processing and communication, except for those with explicit synchronization between nodes. Steps 2, 4, 9, and 11 are also asynchronous in our experiments due to reactive handling of multiple network connections at each endsystem: with a thread-per-connection architecture these steps could be made synchronous, but that architecture may be infeasible in highly connected large-scale sensor networks with stringent real-time and embedded footprint constraints.

3.2 Experimental Platform

All experiments were conducted on a 4-machine cluster of Pentium 4 2.53GHz CPUs, each with 512MB RAM running KURT Linux 2.4.18. In our experiments we used a 10x10 mesh of 100 nodes, a representative mid-scale sensor network topology. To compare scalability of our results, we also ran our experiments with 4 fully connected nodes, each on a separate machine. Finally, we ran the same experiments with 2 nodes, each on its own machine, to study fine-grain communication phasing effects between nodes.

3.3 Comparison Metrics

We used the following metrics to compare the performance of DBA-color using ACE, TAO, and nORB.

Elapsed cycle times: The elapsed time for one cycle of the DBA-color algorithm is the fundamental measurement in our experiments. A node must wait for messages from *all* its neighbors in each cycle of the DBA-color algorithm before it proceeds to the next cycle. Thus, a small

delay in one cycle of a node may be amplified and propagated to its neighbors in the following cycles. This metric’s sensitivity to delay proved very helpful for identifying performance variations between different middleware mechanisms, and motivated various earlier optimizations of nORB [2].

Mechanism-level timing: As Section 3.1 describes and Figure 1 illustrates, timing analysis in distributed concurrent systems such as the DBA-color application must consider both synchronous and asynchronous intervals. In particular, it is essential to measure synchronous intervals to ensure jitter is tightly bounded, and also to detect more egregious problems such as deadlock or large head-of-line blocking effects.

Measuring asynchronous intervals is also important, though it is reasonable to expect the bounds on those intervals to be less strict than the synchronous bounds. In particular, the timing of asynchronous intervals may also shed light on larger-scale performance issues, such as the message buffering issue Section 4.5 describes. We therefore measure time bounds on *each* of the segments enumerated in Section 3.1 and shown in Figure 1, for fine-grain comparison of mechanism-level performance in ACE, TAO and nORB.

3.4 Experimental Methodology

To develop a mechanism-level model for the differences in performance between ACE, TAO, and nORB implementations of the DBA-Color algorithm described in [2], we performed experiments with simple graphs of 2 and 4 nodes respectively, in addition to the full 100-node configuration studied in [2]. We introduced timing *checkpoints* for measuring the time spent on each of the fundamental steps described in Section 3.1, to construct a detailed timing profile of each segment along the end-to-end messaging path for ACE, TAO, and nORB. We used two timers in each node for these experiments:

- an application-level timer to measure the time taken for one application-level cycle, and
- a mechanism-level timer to measure the time taken for the different stages of the middleware layer.

The following paragraphs describe our approach to each of several crucial issues, which we addressed to ensure accuracy and reproducibility of our results.

Application-level Instrumentation: We used each node’s application-level timer to measure each application *cycle* as described in Section 3.1. Immediately before a node started to send messages to its neighbors, its application-level timer was started, along with its mechanism-level timer for the first middleware segment. Specifically, both timers were started in each node just before the *marshal* stage commenced at the beginning of the timeline shown in Figure 1. The application-level timer was stopped and restarted after each complete application cycle. To achieve full timing coverage across all cycles with no intervening gaps, we stopped the application-level timer just before starting it.

Mechanism-level Instrumentation: At each mechanism-level checkpoint, *i.e.*, between each of the segments shown in Figure 1, the mechanism-level timer was stopped, the elapsed time logged in-memory and the timer was started again. As with the application-level timer, we stopped, measured and started the mechanism-level timer contiguously in our measurements, thus ensuring that we did not leave any gaps in the mechanism-level measurements. Hence we ensured reasonably full accounting of the time spent during an entire application-level cycle.

To further verify that our timing measurements offered full coverage, we computed the total time taken for a cycle based on summation of the individual measurements and compared these results to the actual observed cycle time measurements. We verified that those two results matched closely, which indicated that our timing measurements offered reasonably full accounting. We also tagged each of the mechanism-level measurements with a segment identifier, so that related sequences of mechanism-level measurements could be correlated with the overall application-level measurements. This helped immensely in pinpointing the cause of middleware behavior causing higher cycle times in TAO, as described in Section 4.5.

Timer operations were all inline and used pre-allocated memory to avoid instrumentation overhead interfering significantly with actual system performance. Figure 2 in Section 4.2 shows performance of the 100 node configuration with full timer instrumentation, which is nearly indistinguishable from the similar plot without instrumentation originally presented in [2]. This result indicates that

the effect of instrumentation was minimal, as designed.

Scale of Experiments: We conducted identical experiments with the DBA-color application using ACE, TAO, and nORB, and across several different node configurations. The first, using only two nodes on two separate machines was designed to minimize the coupling between nodes, with each node having only a single neighbor and with contention both minimal and limited to only the network resource. The second node configuration used four nodes, each again on its own machine, but with three neighbors each. Finally, we ran the original 10 by 10 hundred node mesh, with 25 nodes running on each of 4 machines as described in Section 3.2. The point of varying the node configurations was to isolate factors of node interaction and resource contention in forming a preliminary performance model for DBA-color running on ACE, TAO, and nORB. To avoid misinterpreting artifacts of network, platform or other experimental noise, we repeated each experiment over roughly 500,000 cycles of the DBA-color algorithm, producing consistent and repeatable distributions of measured timing.

4 Empirical Results

In this section we present our experimental results and analyze them using each of the metrics described in Section 3.3. Section 4.1 first presents a discussion of overall performance results. Section 4.2 then describes observed cycle times. Section 4.3 presents marshaling results. Section 4.4 examines lookup and dispatching performance. Section 4.5 discusses measured asynchronous wait times. Finally, Section 4.6 offers a preliminary performance model based on the other results presented in this section.

4.1 Overall Performance Results

Table 1 shows the mean and median performance values for the application cycle times and individual segments, for each implementation running in 100, 4, or 2 nodes. Of particular interest is that even with the TAO optimization described in Section 4.5, the mean and median application cycle times for nORB are close to that of TAO with nORB performing better than TAO on the average, for higher number of nodes. This effect correlates most

strongly with differences in the mean wait times, which is worse for TAO than nORB for higher number of nodes. Section 4.5 discusses the implications of these wait time results in detail. The performance results show that nORB performs as well as TAO, but with a huge reduction in footprint. With nORB, the code segment size for a statically compiled Node executable (using *size* command) is ~ 567 KB, whereas with TAO it is ~ 1800 KB.

ACE performs better on average (both mean and median) than either nORB or TAO in overall cycle times, marshaling, send, and wait times. TAO’s highly optimized receive mechanism outperforms those for both ACE and nORB. TAO’s lookup and dispatch mechanisms are similarly optimized and outperform nORB: the implementation using ACE does not perform these functions but rather demarshals directly from the socket receive. Finally, demarshaling and algorithm segment times are negligible due to the relative simplicity of these mechanisms. We now turn our attention from average performance values to detailed performance *distributions* in the rest of this section.

4.2 Cycle Times

Figure 2 shows the distribution of measured cycle times over $\sim 500,000$ cycles of the DBA-color algorithm using ACE, nORB and TAO, up to a 25 msec limit that includes 98% of all samples in each case. These measurements

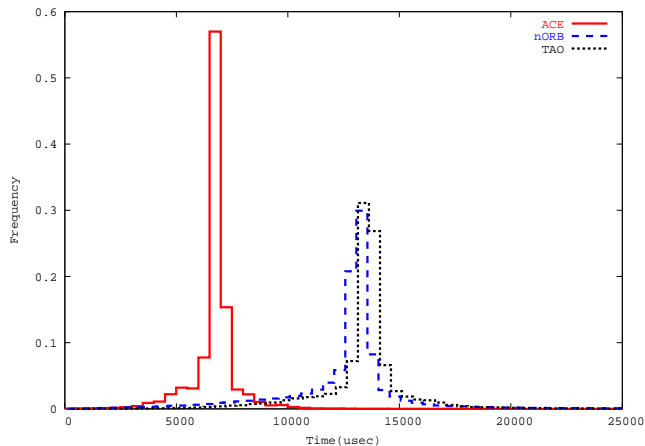


Figure 2: Cycle Times: 100 Nodes

were taken with the additional timing instrumentation in

Configuration	application cycle	marshal	send	wait	receive	lookup and dispatch	demarshal	algorithm segment
ACE: 100 nodes	6928 / 6978	4 / 5	40 / 5	906 / 534	11 / 9	0 / 0	1 / 0	1 / 0
nORB: 100 nodes	13015 / 13210	13 / 10	75 / 7	1703 / 1190	12 / 11	15 / 12	0 / 0	1 / 1
TAO: 100 nodes	13687 / 12667	18 / 9	90 / 7	1764 / 788	9 / 7	12 / 20	0 / 0	1 / 1
ACE: 4 nodes	228 / 227	5 / 3	6 / 3	16 / 16	10 / 9	0 / 0	1 / 0	0 / 0
nORB: 4 nodes	385 / 384	11 / 11	8 / 5	15 / 16	9 / 6	17 / 15	0 / 0	0 / 0
TAO: 4 nodes	411 / 410	11 / 9	8 / 7	29 / 20	9 / 8	7 / 4	0 / 0	0 / 0
ACE: 2 nodes	128 / 128	2 / 2	3 / 1	51 / 52	5 / 5	0 / 0	1 / 1	0 / 0
nORB: 2 nodes	190 / 189	13 / 16	5 / 5	52 / 52	6 / 6	15 / 13	0 / 0	0 / 0
TAO: 2 nodes	165 / 164	12 / 12	5 / 5	50 / 46	7 / 5	4 / 4	0 / 0	0 / 0

Table 1: Mean/Median Timing of Application Cycle and Middleware Mechanisms (in μsec)

place for each of the middleware mechanism segments shown in Table 1. As noted previously in Section 3.4, the shapes and locations of the curves for each of the configurations is similar to those reported in our previous measurements without the fine-grain instrumentation in place [2]. These results thus give evidence that our instrumentation of the middleware had only a limited effect on the performance of the system overall.

Furthermore, the shapes of the distributions are regular and consistent, with a narrower distribution for ACE falling to the left of the wider distributions for nORB and TAO, with the nORB distribution shifted slightly to the left compared to the TAO distribution. The distributions shown in Figure 2 thus reinforce the impressions of overall performance gleaned from Table 1.

4.3 Marshaling

Figure 3 shows the Marshaling times for ACE, TAO, and nORB configurations with 100 nodes. The distributions shown in Figure 3 reinforce the overall impression we get from Table 1, but also show an interesting tail to the right for both nORB and TAO. We can see that the overhead of marshaling a request and writing it to the connection are very similar for nORB and TAO, which is an expected result because both nORB and TAO use ACE’s Common Data Representation (CDR) class to marshal their requests. Marshaling in the ACE implementation of DBA-color outperforms that for both nORB and TAO, for two reasons. First, the ACE implementation only mar-

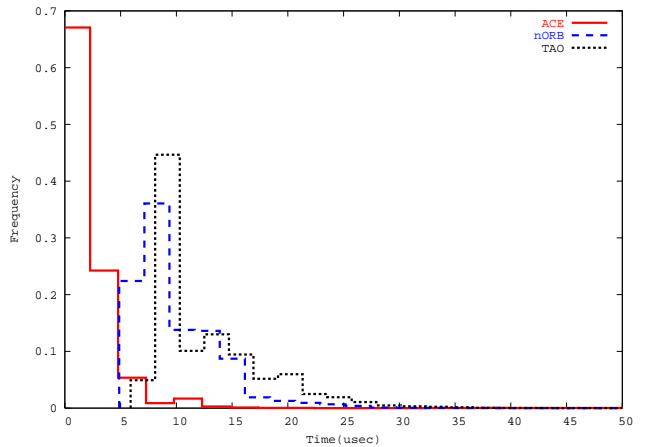


Figure 3: Marshaling Times: 100 Nodes

shals each message once, whereas nORB and TAO marshal both the message header and message body. Second, the ACE implementation only constructs one message, even if it will be sent to multiple neighbors. The results shown in Figure 3 thus correlate strongly with our understanding of the underlying marshaling mechanisms in ACE, TAO, and nORB.

4.4 Lookup and Dispatching

Figure 4 shows the times for servant lookup and method dispatching on the server side in TAO and nORB, running on 4 nodes. We show this case in detail because it

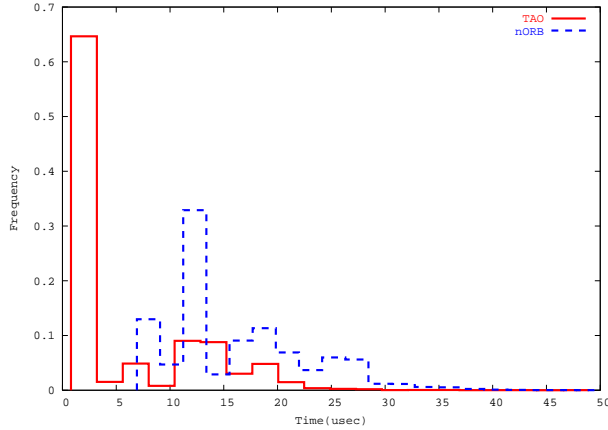


Figure 4: Lookup and Dispatching Times: 4 Nodes

reflected the greatest difference in lookup and dispatching times between nORB and TAO, as shown in Table 1. Times for ACE are not shown in Figure 4, as ACE does not perform lookup or dispatching and the times measured were negligible as expected, which Table 1 also indicates.

4.5 Wait Times

Finally, Figure 5 shows the wait time distributions for TAO with 2-nodes, first with the `SYNC_NONE` messaging policy, and then with the better-suited `SYNC_TRANSPORT` policy. `SYNC_NONE` was originally used by the TAO DBA-color implementation because its use was intended to improve message throughput. However, our experiments showed that for lightly loaded networks with stringent performance requirements, TAO’s default `SYNC_TRANSPORT` policy is strongly preferred. We first identified the need to use the `SYNC_TRANSPORT` policy instead of `SYNC_NONE` in TAO during our performance experiments using 2 nodes. Because we had tagged the individual timing data with ids for the segments being measured, we were able to observe that reasonably frequent cases of ORB-level message buffering were occurring for TAO with 2 nodes. We then examined the code path in TAO and identified the mechanism configured by the `SYNC_NONE` policy as the cause of the observed message delays.

It is of particular interest that although the use of `SYNC_TRANSPORT` significantly reduced long wait times in the implementation using TAO, it did not eliminate

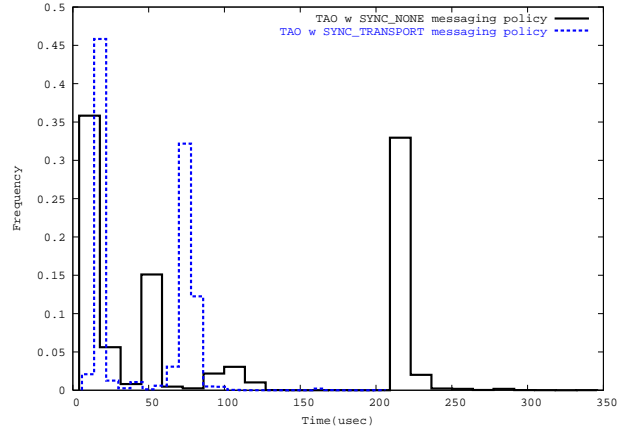


Figure 5: TAO Wait Times: 2 Nodes

them entirely. Combined with the absence of this effect in the implementation using nORB, this suggests the possibility that the reduced message size in nORB may be a contributor to achieving consistently shorter wait times by avoiding multiple sends associated with message fragmentation. This question clearly frames an important next direction in our research.

4.6 A Preliminary Performance Model

From the results in this section, we can infer a reasonably consistent model for the performance observed in [2] and reproduced in the experimental results presented so far in this section. We conclude our analysis of experimental results by summarizing the relative contributions of each kind of middleware segment and offering a plausible explanation for any crucial differences in performance of that segment between ACE, TAO or nORB.

Application Cycle: The time taken by each algorithm cycle affects the total time the algorithm takes to converge. As illustrated in Figure 1, the application cycle stage is influenced by each of the middleware-level stages, as we explain in detail below. Because of the topology of the 100-node graph, a delay in any of these middleware-level stages has a potential ripple-effect, which then may increase the delay experienced in the wait stage and as a result the overall cycle time may be impacted significantly.

Marshal: When a remote call is made, the application data is marshaled to a common on-the-wire format. All three implementations use the CDR format to marshal the data. In the case of ACE, since the data formats are decided at design time, no extra information is needed to identify the destination object in the server or the associated method on the destination object. On the contrary, TAO and nORB need header information to identify the appropriate target object and its method.

Marshaling may cause a significant amount of overhead if performed repeatedly. In the DBA-color application, marshaling may be performed multiple times in one cycle based on the topology of the graph. For example, for the fully connected 4-node graph, this would be done 6 times per cycle at each node: once for each of the 3 neighbors for the color and improvement messages. For TAO and nORB, this amounts to doing the marshaling all 6 times, whereas for ACE, the marshaling can be done once and then the marshaled data sent to all the neighbors. This forms one of the most significant sources of overhead for the Color-DBA implementations using TAO and nORB, when compared to the implementation using ACE.

Send: This stage measures the time incurred to send the byte stream assembled from the previous stage. This is the time it takes to hand over a byte stream buffer from the application buffers to the OS kernel buffers. The length of the byte stream buffer determines the amount of time it takes in this stage. TAO and nORB take more time in this stage compared to ACE because of the header information sent with each request.

Wait: This stage accounts for the idle time in a node, while it is waiting for messages from its neighbors. This time is influenced by the other stages, the distributed nature of the algorithm, and the topology of the graph. The wait time is also sensitive to delays in network. The implementation using ACE shows the least delay in this stage, because of the lower time spent in the other stages.

We observed a potentially interesting property of the wait times in the 2 node TAO configuration using SYNC_NONE. Once a delay was set up, it was possible for the two nodes to lock into a fairly stable and synchronous pattern of message exchange, resulting in persistently long wait times. In the larger graphs this effect was not observed, and we speculate that some elasticity property due to the relative degrees of freedom of

the nodes in the 100 node, 4 node, and 2 node networks is involved. We intend to study these kinds of effects in greater depth as future work.

Receive: Upon receipt of a message byte stream from a client, the server tries to read a header, which contains the total length of the payload. The payload in turn consists of the marshaled request header and application data sent from the client. Based on the information in the header, the request is assembled. In this stage, a very small amount of demarshaling is involved - demarshaling the header information. Since all three implementations do this and the observed times are uniformly small, this stage has little effect on the overall cycle time.

Lookup and Dispatch: Once a request is completely assembled by the previous stage, the request is parsed to dispatch the method to the appropriate server-side implementation object. This involves demarshaling the request header, looking up the servant object using the object key embedded in the request header, looking up the method to be called on the servant object and then making the upcall on to the skeleton object, which finally dispatches the data to the implementation.

The Color-DBA ACE implementation knows the target object of the incoming call at design time and hence does not go through the stage of lookup and dispatch, whereas in TAO and nORB this has to be done for each and every remote call on the server. This is a significant source of overhead and is one of the reasons for the lower cycle times observed for the implementation using ACE. For one remote call, this might not pose a significant overhead. But, as shown in Figure 1, this happens two times the number of neighbors in each cycle on each node. Furthermore, this effect increases the delay on each node as it waits to get data from its neighbors.

Demarshal: The time spent in this stage is the time taken by the skeleton to demarshal the application data payload from the incoming CDR stream, which contains the marshaled payload sent by the client. This does not include the time taken to demarshal the header and request header, since that time is included in the *Lookup and Dispatch* stage. Since the ACE, TAO and nORB implementations use the ACE CDR stream classes, the time taken to demarshal is the same for all of them, since the application message structure is the same across all the

three implementations. We observed that the time taken is very small (sub- μ sec) and hence has very little effect on the overall performance.

Algorithm Segment: After the target has been identified, the method encoded in the request is called on that object. This stage is solely determined by the application logic. In the Color-DBA application, each node evaluates the current color assignments locally and searches for a better assignment which would reduce the amount of violations in the coloring rules. The steps performed in this stage are both simple and exactly the same for the three implementations, and hence have relatively little effect on the overall performance.

5 Related Work

In this section we note special purpose middleware projects that address similar challenges to those in our work, and describe related work on fine-grain middleware performance evaluation.

Embedded and Real-Time Middleware: As described in our previous work [2], three main projects are closely related to our work on nORB: MicroQoS CORBA, Ubiquitous CORBA, and e*ORB. MicroQoS CORBA is a middleware research project at Washington State University, focusing on middleware footprint reduction through case-tool customization of middleware features for highly constrained embedded systems.

Ubiquitous CORBA projects such as LegORB and the CORBA specialization of the Universally Interoperable Core (UIC) focus on a metaprogramming approach to DOC middleware. The key difference with our work is that the UIC contains *meta-level* abstractions that differ from middleware paradigms, *e.g.*, CORBA, must specialize, while ACE, TAO, and nORB are concrete *base-level* frameworks.

e*ORB is a commercial CORBA ORB developed for embedded systems, especially in the Telecom domain. Although the e*ORB web pages claim that e*ORB is the smallest and fastest CORBA ORB, they do not show the kinds of detailed performance comparisons, *in the context of a specific application*, as we have presented here. We plan to expand on the set of fine-grained empirical comparisons presented in this paper, to include e*ORB in

our future studies of special purpose middleware performance.

Middleware Performance Measurement: The work most strongly related to that presented in this paper is the original series of projects measuring, modeling, and optimizing performance in TAO and other CORBA ORBs. For example, detailed studies of IIOP protocol performance and TAO's optimized protocol engine are described in [9]. We leveraged the ACE CDR classes and other optimized mechanisms employed by TAO for nORB, and conducted the similarly detailed experiments reported here, to ensure nORB achieved comparable performance to TAO while offering a reduced ORB footprint.

nORB's concurrency, locking, and memory architecture is very simple due to the domain for which it was designed. While real-time performance is a key concern, the ORB core experiments here covered most of the space of current design concerns. However, as we add planned features such as hybrid static/dynamic scheduling, we will need to conduct further experiments to ensure appropriate real-time concurrency, memory management, and locking patterns are applied as in TAO [10].

6 Conclusions

In this paper we have shown that the development of special purpose middleware requires careful observation and analysis of experimental results during the development process. We have presented a methodology for combined application and middleware mechanism-level performance analysis. Special optimizations such as reduced message sizes may not be achievable through purely COTS standard middleware, but may be available in particular implementations. In addition, discovering *which* settings and features are best for an application requires both careful design *a priori*, followed by empirical measurement and possible design refactoring. It is therefore important to adopt an iterative approach to middleware development that takes current application requirements and experimental results into consideration. In this paper we have thus further emphasized the importance of careful empirical measurement within the context of a representative application, *as an essential tool for the development of special purpose middleware itself*.

The rest of this section is structured as follows. Section 6.1 presents key observations stemming from our results, and makes recommendations to leverage those observations. Section 6.2 then describes future work to follow up on new questions raised in this work.

6.1 Observations and Recommendations

Observation: Redundant Marshaling – A node sends the same application level messages to its neighbors. Using a standard ORB data path causes the same data to be marshaled repeatedly, which is not only unnecessary but also increases the application cycle time, adversely affecting the timeliness of the application. The DBA-color implementation using ACE uses application-level knowledge and optimizes this.

Recommendation: Optimize Marshaling – Using application knowledge, an ORB could be configured to pre-marshall data using some form of memoization at the stub level. Another option would be to apply the principle of group communication and use a group Inter-ORB protocol to achieve this. This would be very useful in sensor network applications where neighborhood communication is the highly preferred and sometimes only available mode of communication.

Observation: Time Complexity for Lookup – TAO performs better than nORB and provides bounded behavior using strategies like perfect hashing for operation lookup. Currently, nORB uses a linear search for operation lookup. This was done based on the assumption that there are only a few objects registered on the server expecting remote calls. But this search strategy could adversely affect delay sensitive applications like Color-DBA, which exhibits a ripple-effect in the overall cycle time.

Recommendation: Use Hashing Techniques – Using hash-based searching for operations is recommended, since this makes the lookup more bounded.

Observation: Large Header Size – The size of the header plays a significant role in the time it takes to marshal and demarshal. In applications where the payload size is small, this could become a bottleneck and alternatives such as payload aggregation or header minimization should be pursued.

Recommendation: Minimize Header Fields – Fine tuning the contents of the header based on the application may be necessary in some embedded systems. This technique can significantly optimize the marshaling, unmarshaling and sending of messages between endsystems, especially when message traffic is heavy. This is an example of the type of application specific customization of generic middleware at the cost of interoperability that can be required for the type of special purpose middleware that is the focus of this research.

6.2 Future Work

We plan to conduct the following areas of future research that were suggested by the results presented in this paper.

Marshaling Optimization: There are several different candidate mechanisms to avoid redundant marshaling when the same data is sent to different neighbors. Of these, we are planning to investigate *memoization* (an algorithmic technique which saves a computed answer for later reuse, rather than recomputing the answer), multicast, and Group IORs.

Communication Models: We plan to investigate communication models other than the oneway communication model that we have used in our experiments here, for example using the TAO Event Service and AMI framework. It would be useful to analyze how the application properties affect the performance with each of these additional inter-node communication models.

Operation Lookup: Relatively slower operation dispatches were observed for nORB in our experimental results, which we would like investigate further. We will try to identify the causes of this, and mitigate them to reduce temporal overhead as much as possible without incurring excessive spatial overhead or programming model complexity.

Configurability: We plan to investigate further the relationships between timeliness, footprint, and feature sets in nORB. Ideally, it should be possible to configure the ORB selectively at design time as well as run-time, to achieve maximal performance in terms of both time and space.

Jitter: From Table 1, we infer that the jitter for the send stage is relatively higher for the 100 node experiment than

for the other two. We plan to investigate the cause of this jitter.

7 Acknowledgements

We gratefully acknowledge the support and guidance of the Boeing NEST OEP Principal Investigator Dr. Kirby Keller and Middleware Principal Investigator Dr. Doug Stuart. We also wish to thank Dr. Weixong Zhang at Washington University in St. Louis for providing the initial algorithm implementation used in DBA-color.

References

- [1] D. Corman, "WSOA-Weapon Systems Open Architecture Demonstration-Using Emerging Open System Architecture Standards to Enable Innovative Techniques for Time Critical Target (TCT) Prosecution," in *Proceedings of the 20th IEEE/AIAA Digital Avionics Systems Conference (DASC)*, Oct. 2001.
- [2] V. Subramonian, G. Xing, C. Gill, and R. Cytron, "The Design and Performance of Special Purpose Middleware: A Sensor Networks Case Study," *Submitted to the 9th IEEE Real-Time Technology and Applications Symposium*, May 2003.
- [3] D. C. Schmidt and et al., "TAO: A Pattern-Oriented Object Request Broker for Distributed Real-time and Embedded Systems," *IEEE Distributed Systems Online*, vol. 3, Feb. 2002.
- [4] D. Estrin, D. Culler, K. Pister, and G. Sukhatme, "Connecting the Physical World with Pervasive Networks," *IEEE Pervasive Computing*, vol. 1, Mar. 2002.
- [5] W. R. Heinzelman, A. Chandrakasan, and H. Balakrishnan, "Energy-efficient communication protocol for wireless microsensor networks," in *HICSS*, 2000.
- [6] W. Zhang, G. Wang, and L. Wittenburg, "Distributed stochastic search for constraint satisfaction and optimization: Parallelism, phase transitions and performance," in *Proceedings of AAAI Workshop on Probabilistic Approaches in Search*, 2002.
- [7] S. Minton, M. D. Jonston, A. B. Philips, and P. Laird, "Minimizing conflicts: a heuristic repair method for constraint satisfaction and scheduling problems," *Artificial Intelligence*, vol. 58, pp. 161–205, 1992.
- [8] M. Yokoo, E. H. Durfee, T. Ishida, and K. Kuwabara, "Distributed constraint satisfaction for formalizing distributed problem solving," in *International Conference on Distributed Computing Systems*, pp. 614–621, 1992.
- [9] A. Gokhale and D. C. Schmidt, "Optimizing a CORBA IIOP Protocol Engine for Minimal Footprint Multimedia Systems," *Journal on Selected Areas in Communications special issue on Service Enabling Platforms for Networked Multimedia Systems*, vol. 17, Sept. 1999.
- [10] I. Pyarali, D. C. Schmidt, and R. Cytron, "Techniques for Enhancing Real-time CORBA Quality of Service," *IEEE Proceedings Special Issue on Real-time Systems*, May 2003.