Middleware Specialization for Memory-Constrained Networked Embedded Systems *

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

Abstract

General purpose middleware has been shown to be effective off-the-shelf, in meeting diverse functional requirements for a wide range of distributed systems. However, middleware customization is necessary for many networked embedded systems because of the resource constraints in the networked nodes. In this paper, we demonstrate that reduced middleware footprint can be achieved while maintaining real-time properties of applications running on such systems. We also give evidence that empirical measurement using a representative application is crucial to guide (1) selection of feature subsets from general purpose middleware and (2) trade-offs among different dimensions of design metrics including real-time, footprint, and portability.

1. Introduction

Different kinds of general purpose middleware have become key enabling technologies for many 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 in large-scale applications [1]. However, simply adding features and layers is ill-suited for certain kinds of applications. In particular, features are rarely innocuous in applications with requirements for both real-time performance and small memory footprint. Instead, every feature is likely to either contribute to or detract from the application in one or both of those dimensions. It is thus crucial to study each feature's advantages and disadvantages carefully.

For networked embedded 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 [11], a widely used real-time CORBA Object Request Broker (ORB) strikes a balance between these two design forces, with performance optimization and real-time assurance mechanisms provided as first-class features. Similarly, in this paper we show that nORB, a specialized ORB for networked real-time embedded systems developed at Washington University, achieves real-time performance that is similar to TAO, while reducing footprint significantly.

This paper is structured as follows. Section 2 describes an example networked embedded application. Section 3 lists the design issues and tradeoffs that we encountered while designing and developing special purpose middleware. Section 4 describes the design of our experiments to quantify application and middleware performance in a realistic setting. Section 5 presents the results of our experiments, and offers analysis of those results. Section 6 describes related work on special-purpose middleware. Finally, Section 7 offers concluding remarks and describes future work.

2. Memory-Constrained Networked Embedded Systems

Distributed networked embedded systems are being used in a variety of different applications ranging from temperature monitoring to battlefield strategy planning [2]. 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. Networked embedded systems thus challenge classical approaches to distributed computing and represent an active research area with many open questions. This section introduces a real-world networked embedded systems problem and distributed algorithmic approaches that have been applied to solving that problem.

2.1. Example: Active Damage Detection

In this application, a number of MEMS sensor/actuator nodes are mounted on the surface of a physical structure

*This work was supported in part by the DARPA NEST (contract F33615-01-C-1898) and PCES (contract F33615-00-C-1697) programs.

such as an aircraft wing. Such a structure may be damaged during operation, and the goal of this application is to detect such damage when it occurs. Vibration sensor/actuator nodes are arranged in a mesh with (wired or wireless) network connectivity to a fixed number of neighboring nodes. To detect possible damage, selected actuators called *ping nodes* generate vibrations which propagate across the surface of the physical structure. Sensors within a defined neighborhood can then detect possible damage near their locations by measuring the frequencies and strengths of these induced vibrations. The sensors convey their data periodically to central nodes in the system, which process the data to detect any damage and can issue alerts or initiate corrective actions accordingly.

Ping Node Scheduling: Three restrictions on the system make the problem of damage detection difficult. First, the sensor/actuator nodes are resource-constrained; second, two vibrations whose strengths are above a certain threshold at a sensor location will interfere with each other; and third, sensor/actuator nodes may malfunction over time. These constraints, therefore, require that the actions of two overlapping ping nodes be synchronized so that no interfering vibrations will be generated at a sensor location at any time.

This damage detection problem can be captured by a constraint model. Scheduling the activities of the ping nodes can be formulated as a distributed graph coloring problem. A color corresponds to a specific time slot in which a ping node vibrates. Thus two adjacent nodes in the graph, each representing an actuator, cannot have the same color since the vibrations from these actuators would then interfere with each other. The number of colors is therefore the length (in distinct time slots) of a schedule. The problem is to find a shortest schedule such that the ping nodes do not interfere with one another, in order to minimize damage detection and response times. Distributed algorithms [16] have been shown to be effective for solving the distributed constraint satisfaction problem in such large scale and dynamic (*e.g.*, with occasional reconfiguration due to sensor/actuator failures on-line) networks.

2.2. Middleware Challenges

The application described in Section 2.1 is representative of a broader class of networked embedded systems in which (1) a large number of networked sensor/actuator nodes are networked together and (2) those nodes collaborate to perform a common task under a defined set of constraints. A system implementation is needed to facilitate the exchanges of local information between nodes. To reduce implementation costs across families of similar networked embedded systems, the implementation must also be re-usable across a variety of different distributed constraint satisfaction applications, network topologies, and node features. Specifically, a middleware framework that abstracts common services like communication is needed. The key challenges in the design and implementation of such middleware are to:

Reuse Existing Infrastructure: We want to avoid developing new middleware from scratch. Rather, we want to reuse pre-built infrastructure to the extent possible.

Provide Real-time Assurances: Middleware itself must be predictable to allow application-level predictability.

Provide Robust Distributed Object Computing (DOC): We chose the DOC communication paradigm since it offers a more maintainable programming model and allows direct communication among remote and local components, thus increasing location independence.

Reduce Middleware Footprint: The target environment for this middleware is the sensor/actuator nodes, which have constraints on the amount of RAM/ROM on board.

3. Design Trade-offs in Networked Embedded Systems Middleware

In this section we describe the design issues and trade-offs that we encountered in designing and developing special purpose middleware to meet the challenges described in Section 2.2. While we focus specifically on our work on nORB, the same methodology can be applied to produce middleware tailored to other special purpose applications.

3.1. Development Philosophy

Standard CORBA implementations like TAO [11] and e*ORB [6] offer generic CORBA implementations, whose feature sets are determined *a priori*. ORBs like e*ORB that offer small feature sets are likely to have fewer unneeded features, but even there the feature set is not tailored to the exact requirements of a particular application. Furthermore, faithful implementation of the entire CORBA standard increases the number of features supported by these ORBs and hence results in increased footprint for the application. In the case of memory-constrained networked embedded applications, this can become prohibitively expensive.

We instead use a bottom-up composition approach to get *only* the features that we need. The selection of features for our special purpose middleware implementation was strictly driven by the unique requirements of the application domain. Due to space limitations, we refer the interested reader to [3] for more information on the bottom up compositional approach that we used to develop nORB.

3.2. Limited Data Type Support

We used the application domain to guide our choice of data types that the ORB middleware supports. In the damage detection application, for example, sequences of simple structures are exchanged between sensor/actuator nodes. nORB therefore supports only basic data types, structures and sequences. This reduces the code base to support data types to the bare minimum necessary for the ping node scheduling application. We choose to keep marshaling in nORB to support application deployment over heterogeneous networked embedded platforms. For other application domains with homogeneous platforms, support for marshaling could be removed entirely.

Trade-off: To support other data types, the middleware has to incorporate the code to marshal and unmarshal these data types properly, thus increasing footprint. At a cost of footprint, we could support more advanced data types like CORBA *Any*, if an application required them.

3.3. Simplified Messaging Protocols

nORB provides protocols that enable communication between end-system components, while meeting the footprint reduction challenge described in Section 2.2. Such protocols define both the sequence and format of messages exchanged between end-systems. Previous work [7] has shown that optimizations can be achieved by the principle patterns of 1) relaxing system requirements and 2) avoiding unnecessary generality. Protocols in TAO like *GIOP-Lite* [7] are designed according to this principle. Similarly, we support a limited subset of the message types in the CORBA specification, so that we incur only the necessary footprint, while providing all features required by the application. Specifically, nORB supports Request, Reply, Locate Request, and Locate Reply message types. The format of the Request and Reply messages closely resembles that of the GIOP Request and Reply messages respectively, the major difference being the elimination of the service context field in the message headers.

Trade-off: By reducing the number of fields in the header, advanced features of a CORBA ORB such as Portable Interceptors are not supported. Providing this support would increase both ORB and message footprint.

3.4. Simplified Life-cycle Management

In a networked embedded system, the number of application objects hosted on each node is expected to be very small, which reduces the need for full-fledged life cycle management. Servant objects are registered when the application starts, and live as long as the application, eliminating the need for more complicated dynamic life-cycle management. Even though the resulting object adapter in nORB does not conform to the CORBA Portable Object Adapter specification, a significant footprint reduction is achieved due to reduced object adapter functionality. We also consolidated object registration with other setup functions, by moving it from the object adapter interface to the ORB interface.

Trade-off: For large number of objects, or where objects are dynamically created and destroyed, simplified object life-cycle management won't be sufficient. For an end-system that uses a large number of objects, more complex adapters are necessary, albeit at a cost of larger footprint.

3.5. Simplified Operation Lookup and Dispatch

When a remote operation request is received on a server, an ORB must search for the appropriate object in its Object Adapter and then perform a lookup of the appropriate method in the operations table. nORB uses linear search for that lookup because of the assumption that only a few objects on each node expect remote calls, to only a few methods on those objects. The linear search strategy reduces memory footprint while still maintaining real-time properties for small numbers of objects.

Trade-off: For large number of operations, real-time performance will suffer. This is because linear search will take O(n) time to do a lookup, where n is the number of operations. Perfect hashing takes O(1) time to do the actual lookup. This alternative would again entail increased footprint due to the code generated to support perfect hashing.

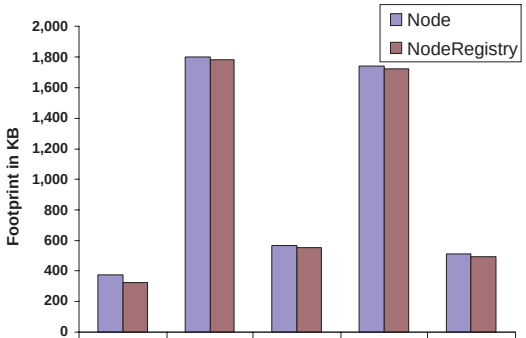
3.6. Message Flow Architecture

Our messaging and concurrency architecture is based on previous work [9, 12, 10] done in TAO. When a client makes a remote *two-way* function call, the caller thread needs to block until it receives a reply back from the server. The two-way function call is made on the client stub, which then marshals the parameters into a Request and sends it to the server. There are different strategies to wait on the client side for the reply, of which we have chosen the *Wait on Connection* strategy for implementation in nORB. On the server side, to process an incoming request and send the reply back to the client, we chose the *Direct Upcall Strategy* for nORB. With this strategy, the servant upcall is made in the context of the network I/O thread, to improve real-time predictability.

Trade-off: With nested upcalls, the *Wait on Connection* strategy could result in deadlocks [14]. The *Wait on Reactor* strategy, on the other hand, avoids deadlocks, but introduces blocking factors which could hurt real-time performance [14].

3.7. Resulting Footprint Reductions

Figure 1 shows the footprint reductions achieved by our approach. All measurements were taken using the *size* command. Node and NodeRegistry are software agents used by the ping scheduling algorithm in the damage detection application. Application specific code in Node and NodeRegistry takes about 164KB and 146KB respectively. Our footprint measurements show that ACE introduces an overhead of ∼212KB, and unoptimized versions of TAO and nORB introduce an additional overhead of 1424KB and 191KB respectively. Compiler optimization of TAO and nORB reduces the added overhead of the ORB layer to 1362KB and 133KB respectively. Thus, the compile-optimized version of nORB achieves a slightly better than 90% reduction in footprint compared to TAO.



	ACE	TAO	nORB	compile optimized TAO	compile optimized nORB
Node	376	1800	567	1738	509
NodeRegistry	324	1778	549	1725	492

Figure 1. Ping Scheduling Footprint Comparison

Although the code size of the nORB-based implementation is 35% larger than that of the ACE-based version, DOC middleware such as nORB provides an easier and more intuitive programming model, abstracting away low-level concurrency and communication details. In using ACE, we made and corrected several implementation errors that nORB avoided through embodiment of canonical middleware design patterns [13]. The ACE-based implementation's primary performace enhancement is the elimination of multiple marshalling (see section 5.5) which we are adding to nORB as future work. Moreover, performance limitations resulting from potential mis-application of strategic design patterns [13] can be avoided through use of middleware that applies those patterns appropriately.

4. Experimental Evaluation

In this section we describe a set of experiments conducted to quantify fine-grain middleware performance of nORB in comparison with ACE and TAO.

4.1. Application Segments

Figure 2 illustrates the essential segments of each of two interacting processes running the ping scheduling algorithm, and shows the messages exchanged between them.

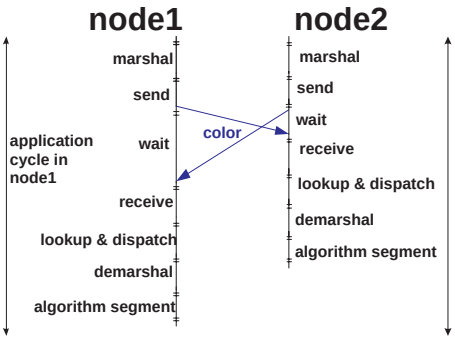


Figure 2. Detailed Ping Scheduling Algorithm Cycle

Each process performs the following sequence of steps in each ping scheduling algorithm execution 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. unmarshals the color value from each message,
7. (algorithm segment) decides its best improve value,
8. Execute above steps once more exchanging *improve* value messages instead of *color* value messages and decide the new color value at the end.

Figure 2 highlights the fact that while some steps are synchronous within a process, *i.e.*, steps 1, 5-7, other steps are asynchronous, *i.e.*, steps 2-4. Furthermore, as Figure 2 illustrates this asynchrony can lead to variations in cycle times, both between and within nodes.

Step 3 is 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 and 4 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 networked embedded systems with stringent real-time and embedded footprint constraints.

4.2. 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 10×10 mesh of 100 nodes, a representative mid-scale networked embedded system topology. To examine 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.

4.3. Comparison Metrics

Elapsed Cycle Times: The elapsed time for one cycle of the ping scheduling algorithm is the fundamental measurement in our experiments. A node must wait for messages from *all* its neighbors in each cycle of the ping scheduling 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 helpful for identifying performance variations between different middleware mechanisms, and motivated many of our optimizations to nORB.

Mechanism-level Timing: As Section 4.1 describes and Figure 2 illustrates, timing analysis in distributed concurrent systems such as the damage detection 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 that we plan to investigate as part of our future work. We therefore measure time bounds on *each* of the segments enumerated in Section 4.1 and shown in Figure 2, for fine-grain comparison of mechanism-level performance in ACE, TAO and nORB.

5. Empirical Results

In this section we present our experimental results and analyze them using each of the metrics described in Section 4.3.

5.1. 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 appropriate TAO optimizations the mean and median application cycle times for nORB are close to that of TAO, with nORB performing better than TAO on average for higher numbers of nodes. This effect correlates most strongly with differences in the mean wait times, which is worse for TAO than nORB for higher numbers of nodes. The performance results show that nORB performs as well as TAO, but with a significant reduction in footprint. With nORB, the code segment size for a statically compiled Node executable (using the *size* command) is $\sim$567KB, whereas with TAO it is $\sim$1800KB.

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 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 unmarshals directly from the socket receive. Finally, unmarshaling 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.

5.2. Cycle Times

Figure 3 shows the distribution of measured cycle times over $\sim$500,000 cycles of the ping scheduling algorithm using ACE, nORB and TAO, up to a 25 msec limit that includes 98% of all samples in each case. These measurements were taken with the additional timing instrumentation in place for each of the middleware mechanism segments shown in Table 1.

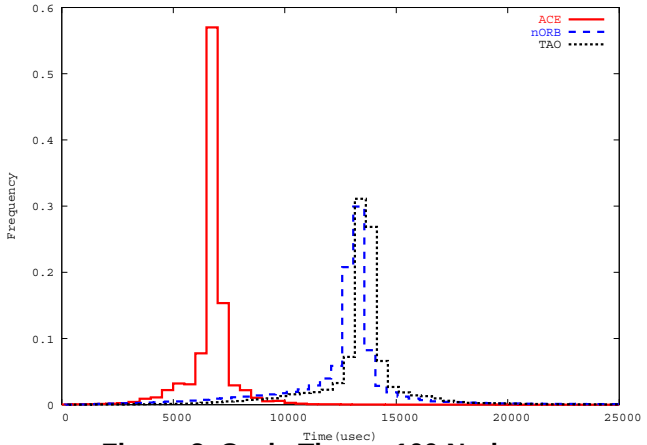


Figure 3. Cycle Times: 100 Nodes

The distributions are regular and consistent, with a narrower distribution for ACE falling to the left of the wider distributions for nORB and TAO, and the nORB distribution

Configuration	application cycle	marshal	send	wait	receive	lookup and dispatch	unmarshal	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)

shifted slightly to the left compared to the TAO distribution. The distributions shown in Figure 3 thus reinforce the impressions of overall performance gleaned from Table 1.

5.3. Marshaling

Figure 4 shows the Marshaling times for ACE, TAO, and nORB configurations with 100 nodes. The distributions shown in Figure 4 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 outperforms that for both nORB and TAO, for two reasons. First, the ACE implementation only marshals 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. In nORB and TAO, marshaling each message is necessary to maintain single message delivery semantics mandated by the CORBA standard. The results shown in Figure 4 thus correlate strongly with our understanding of the underlying marshaling mechanisms in ACE, TAO, and nORB.

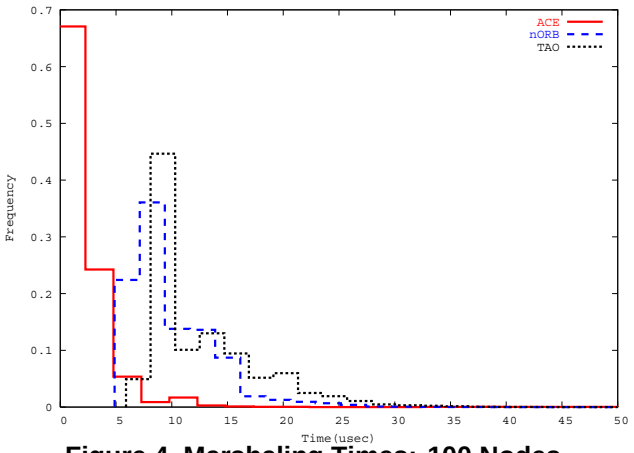


Figure 4. Marshaling Times: 100 Nodes

5.4. Lookup and Dispatching

Figure 5 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 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 5, as ACE does not perform lookup or dispatching and the times measured were negligible as expected, which Table 1 also indicates.

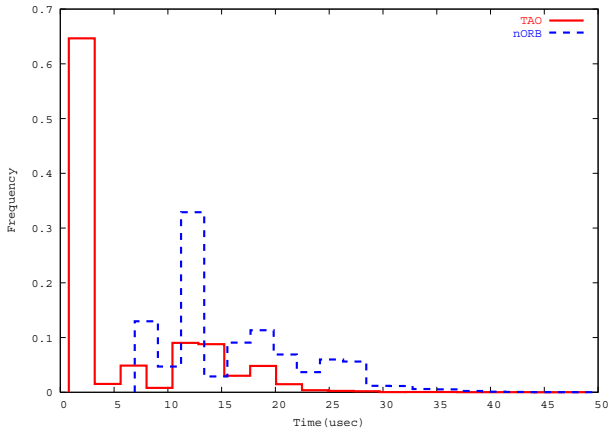


Figure 5. Lookup and Dispatching Times: 4 Nodes

5.5. Analysis of Experimental Results

We now summarize the relative contributions of each middleware segment and offer 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 2, 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 impact the overall cycle time significantly.

Marshal: Marshaling may cause a significant amount of overhead if performed repeatedly. In the ping scheduling implementation, 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. Conversely for ACE, the marshaling can be done only 2 times, once for each of the message, with each marshaled message sent to all the neighbors. This constitutes one of the most significant sources of overhead for the ping scheduling algorithm implementations using TAO and nORB, when compared to the implementation using ACE.

Send: This stage measures the cost to send the byte stream assembled in the previous stage. This is the time it takes to hand over a byte stream buffer from the application to the OS kernel. 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 the network. The implementation using ACE shows the least delay in this stage, presumably because of the lower time spent in the other stages. We surmise that the median wait time for the TAO-based implementation is less than that of nORB for the same reason. 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 that 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 header information, the request is assembled. In this stage, very little unmarshaling is involved - only for the header. 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 unmarshaling 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.

The implementation using ACE knows the target object of the incoming call at design time and hence does not go through the lookup and dispatch stages, whereas in TAO and nORB this is done for 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 2, 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.

Unmarshal: The time spent in this stage is the time taken by the skeleton to unmarshal 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 unmarshal 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 unmarshal 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-μsec) and hence has very little effect on the overall performance.

Algorithm Segment: After the target is identified, the method encoded in the request is called on that object. This stage is solely determined by the application logic. In the ping scheduling algorithm implementation, each node evaluates the current color assignments locally and searches for a better assignment which would reduce the number of violations of 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.

6. Related Work

MicroQoSCORBA [4] at Washington State University focuses on footprint reduction through case-tool customization of middleware features. Ubiquitous CORBA projects [8] 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 different middleware paradigms, *e.g.*,

CORBA, must specialize, while ACE, TAO, and nORB are concrete *base-level* frameworks. e*ORB [6] 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.

Scout [5] is a configurable, communication-oriented operating system targeted at network appliances, such as network-attached devices, multimedia workstations, and network servers. The Ensemble [15] system offers tools that assist the application in guaranteeing properties like reliability and security even in settings where failures or dynamic reconfiguration may be needed or where the network may be under some form of attack.

7. Concluding Remarks

In this paper we have shown that meeting the constraints of networked embedded systems requires careful empirical measurement and analysis within the context of a representative application, *as an essential tool for the development of special purpose middleware itself*. 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. We empirically verified that with nORB[1] we achieved a 90% reduction in footprint over TAO while still retaining real time properties similar to that of TAO.

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 ping scheduling.

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] D. Estrin, D. Culler, K. Pister, and G. Sukhatme. Connecting the Physical World with Pervasive Networks. *IEEE Pervsive Computing*, 1(1), Mar. 2002.

[3] C. Gill, V. Subramonian, J. Parsons, H.-M. Huang, S. Torri, D. Niehaus, and D. Stuart. ORB Middleware Evolution for Networked Embedded Systems. In *Proceedings of the 8th International Workshop on Object Oriented Real-time Dependable Systems (WORDS'03)*, Guadalajara, Mexico, Jan. 2003.

[4] D. McKinnon, D. Bakken, and et al. A Configurable Middleware Framework with Multiple Quality of Service Properties for Small Embedded Systems. In *2nd IEEE International Symposium on Network Computing and Applications*. IEEE, Apr. 2003.

[5] A. Montz, D. Mosberger, S. W. O'Malley, L. Peterson, and T. Proebsting. Scout: A Communications-Oriented Operating System. In *5th Workshop on Hot Topics in Operating System*. IEEE Computer Society Press, May 1995.

[6] PrismTech. eORB. www.prismtechnologies.com/English/Products/CORBA/.

[7] I. Pyarali, C. O'Ryan, D. C. Schmidt, N. Wang, V. Kachroo, and A. Gokhale. Using Principle Patterns to Optimize Real-time ORBs. *IEEE Concurrency Magazine*, 8(1), 2000.

[8] M. Román, R. H. Campbell, and F. Kon. Reflective middleware: From your desk to your hand. *IEEE Distributed Systems Online*, 2(5), 2001.

[9] D. C. Schmidt and C. Cleeland. Applying Patterns to Develop Extensible ORB Middleware. *IEEE Communications Magazine*, 37(4), Apr. 1999.

[10] D. C. Schmidt and C. Cleeland. Applying a Pattern Language to Develop Extensible ORB Middleware. In L. Rising, editor, *Design Patterns in Communications*. Cambridge University Press, 2000.

[11] D. C. Schmidt and et al. TAO: A Pattern-Oriented Object Request Broker for Distributed Real-time and Embedded Systems. *IEEE Distributed Systems Online*, 3(2), Feb. 2002.

[12] D. C. Schmidt, D. L. Levine, and C. Cleeland. Architectures and Patterns for Developing High-performance, Real-time ORB Endsystems. In *Advances in Computers*. Academic Press, 1999.

[13] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, Volume 2*. Wiley & Sons, New York, 2000.

[14] V. Subramonian and C. Gill. A Generative Programming Framework for Adaptive Middleware. In *Hawaii International Conference on System Sciences, Software Technology Track, Adaptive and Evolvable Software Systems Minitrack, HICSS 2003*, Honolulu, HW, Jan. 2003. HICSS.

[15] R. van Renesse, K. Birman, M. Hayden, A. Vaysburd, and D. Karr. Building adaptive systems using ensemble. *Softw. Pract. Exper.*, 28(9):963–979, 1998.

[16] 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.

[1] nORB is freely available as open-source software at http://deuce.doc.wustl.edu/nORB/