

# Design and Implementation of nORB \*

Venkita Subramonian, Christopher Gill  
{venkita,cdgill}@cs.wustl.edu  
Department of Computer Science  
Washington University, St.Louis

Doug Stuart  
douglas.a.stuart@boeing.com  
The Boeing Company  
St. Louis, MO

## Abstract

*Networked Sensors are finding use in a variety of different applications ranging from temperature monitoring to battlefield strategy planning. These networks challenge many classical approaches to distributed computing and provide a new and exciting research area with many open research questions. Standards-based COTS middleware has been shown to be effective in meeting a range of functional and QoS requirements for distributed real-time and embedded (DRE) systems. Each standard makes limiting assumptions, often implicit, about the fundamental set of system capabilities and constraints typical of the domain to which the standard applies. When the characteristics of a particular class of systems violates a standard's assumptions, it may be appropriate to modify or extend the standard and its conforming implementations to better match the actual characteristics of that class of systems while still exploiting the capabilities of the standard.*

*Networked Embedded Software systems fall under the latter category of systems 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. In this paper, we describe an application using networked sensors and how ORB middleware is used to solve some of the common challenges encountered in such applications. We describe our recent work on nORB, a small footprint ORB middleware framework for the Boeing Open Experimental Platform (OEP) under the DARPA NEST program, to meet this entire set of requirements by adapting, unifying, and extending patterns and techniques from earlier related research on COTS middleware frameworks, such as UBI-core, ACE, Kokyu, and TAO.*

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

## 1 Introduction

A primary goal of the DARPA Networked Embedded Systems Technology (NEST) program is to enable "fine-grain" fusion of physical and information processes [1]. Advances in Micro Electro Mechanical Sensor (MEMS) hardware technology has made possible to move software closer to tiny sensors and make them more intelligent. To meet the above goal, a new class of Distributed Real-time and Embedded (DRE) systems, which we term *NEST-class*, is needed. The hardware infrastructure for NEST-class systems consists of a network of 100 to 10,000 fine grained computing nodes, each closely coupled with local sensors and actuators.

To exploit the capabilities of NEST-class hardware infrastructures, the NEST program seeks to provide [1]:

- application independent time-bounded coordination services that adapt to environmental changes,
- time-bounded synthesis methods for resource reallocation in NEST-class systems,
- automated composition and adaptive run-time customization of coordination services, and
- formal verification and assurance of algorithms, software, and system properties.

To realize these objectives for different problem domains, the NEST program is developing several Open Experimental Platforms (OEPs), each with its own challenge problems, design constraints, and resulting infrastructure properties. The OEPs are aligned along three dimensions - *real time*, *resource constraints* and *scale*. In this paper, we focus on middleware support issues for the Boeing

---

\*This work was funded in part by the DARPA NEST program.



### 3 Related Work

In this section we describe related middleware research projects that address some of the challenges described in Section 4.1. For each project, we identify the key assumptions within which it approaches these challenges, and compare those assumptions to the assumptions of the Boeing NEST OEP domain.

**ACE:** The ADAPTIVE Communication Environment (ACE) framework [3, 4] addresses the challenge of providing a robust and portable DOC middleware framework as it:

- encapsulates syntactic variations between operating system APIs, to provide a consistent and portable POSIX-like system interface at the lowest-level,
- provides higher-level abstractions related to the semantics of system services, *e.g.*, threads and thread priorities, and
- implements key design patterns for programming concurrent and networked objects [5].

ACE thus reduces the complexity of the programming model for writing distributed OO applications and middleware infrastructure, and meets the challenge of reuse [6] of infrastructure, patterns, and techniques. ACE does not directly address the challenges of real-time assurances, time-bounded adaptation, reduced footprint, or interoperability with standards-based middleware, but its modular and pattern-rich design facilitates addressing these issues in higher-level middleware such as the nORB framework described in Section 5.

**Kokyu:** Kokyu [7] is a low-level middleware framework built on ACE, for flexible multi-paradigm scheduling [8] and configurable dispatching of real-time operations. Kokyu abstracts combinations of fundamental real-time scheduling and dispatching elements to enforce a variety of real-time policies, including well-known strategies such as Rate-Monotonic Scheduling (RMS) [9], Earliest Deadline First (EDF) [9], and Maximum Urgency First [10].

Consistent with ACE, Kokyu is a lower-level middleware framework that does not address explicitly the challenge of interoperability with standards-based middleware. Kokyu primarily addresses the challenge of enforcing

real-time properties, but is also modular and configurable by design to facilitate addressing the challenges of time-bounded adaptation and reduced footprint in higher-level middleware.

**TAO:** Standards-based DOC middleware technologies such as CORBA [11] and Java RMI [12] have been used historically to achieve location transparency in distributed applications. However, the standards upon which these technologies are based do not address specification or enforcement of the stringent QoS assurances needed by DRE systems.

Recent and emerging DOC middleware standards such as Real-Time CORBA 1.0 (RT-CORBA) [13], Dynamic Scheduling Real-Time CORBA 2.0 [14], the Real-Time Specification for Java [15] and the Distributed Real-Time Specification for Java [16], have addressed a number of key issues related to QoS assurances for DRE systems. Even so, these standards do not fully specify all issues related to rigorous QoS assurance in DRE systems, especially those related to robust, efficient and reusable implementation patterns and techniques.

Therefore, implementations of these standards have offered extensions that may be suitable for incorporation back into future revisions of the standard, or will at least help users of particular implementations achieve necessary QoS assurances for their systems.

TAO [17, 18] is a widely used standards-compliant (ORB) built using the ACE framework. With its real-time extensions [19, 20], TAO explicitly addresses the challenges of portability, re-use, real-time assurances, and interoperability with standards-based middleware. TAO addresses time-bounded adaptation in conjunction with the Kokyu framework and has been used to support adaptive rescheduling of periodic real-time operations as part of the Weapons System Open Architecture (WSOA) [21] program. Finally, footprint reduction has been addressed in TAO, particularly through feature subsetting beyond the level of the Minimum CORBA standard [22], for which TAO can also be configured. However, as we discuss in Section ??, the level of subsetting in TAO motivates our choice of a bottom-up compositional approach starting from the ACE level.

**MicroQoS CORBA:** MicroQoS CORBA [23] is a middleware research project at Washington State University, focusing on extreme middleware footprint reduction [24]

through customization [25] of middleware features for deeply embedded systems.

Where ACE, Kokyu, and TAO focus on framework-oriented middleware infrastructures, MicroQoS CORBA takes a CASE-tool approach using customized IDL compilation, to

- generate client stubs and server skeletons supporting only the data and exception types used,
- choose specific transports and protocols,
- configure low-level marshaling properties, *e.g.*, endian compatibility and inlining, and
- select ORB infrastructure supporting feasible combinations of the QoS features needed.

MicroQoS CORBA clearly addresses the challenges of footprint reduction and interoperation with standards-based middleware. It also addresses the challenges of re-use and of providing a robust and portable middleware framework, but does so by *generating* particular instantiations rather than providing a common infrastructure framework. It addresses real-time QoS assurances by providing higher-level hooks for selecting appropriate ORB infrastructure elements, but does not define explicitly the lower-level policies and mechanisms to achieve those assurances.

Our solution differs from MicroQoS CORBA in that we view frameworks as essential to capture inherent structure of the middleware solution space, *e.g.*, the pluggable transports infrastructure. We share the view of MicroQoS CORBA that explicit representation and composition of features within the *variable* portions of the solution space is essential to meeting stringent footprint and timeliness requirements. However, while the MicroQoS CORBA approach appears to support arbitrary specification of features, it is not clear whether the CASE tool designer or the developer using the CASE tool is responsible for capturing such inherent structure and thus simplifying the complex task of configuring wide ranges of features.

**Ubiquitous CORBA:** Where MicroQoS CORBA focuses on framework-independent generation of ORB infrastructure, related Ubiquitous CORBA projects such as LegORB [26] and the CORBA specialization [27] of

the minimal Universally Interoperable Core (UIC) [28] focuses on a metaprogramming framework approach to DOC middleware.

Both the MicroQoS CORBA and Ubiquitous CORBA approaches involve bottom-up production of an infrastructure consisting only of needed features. However, the Ubiquitous CORBA approach exhibits a significant similarity to the ACE, Kokyu, and TAO approaches in that it assumes a general set of primitives and a framework within which those primitives are arranged. The key difference between the Ubiquitous CORBA approach and the ACE, Kokyu, and TAO approaches is that the UIC contains *meta-level* abstractions that different middleware paradigms, *e.g.*, CORBA or SOAP [29], must specialize [27], while ACE, Kokyu, and TAO are concrete *base-level* frameworks.

The Ubiquitous CORBA approach supports robust portability even across different DOC middleware paradigms. It offers significant re-use of infrastructure, patterns, and techniques by generalizing features common to multiple DOC middleware paradigms and providing them within a minimal metaprogramming framework, thus also addressing the challenge of reducing middleware footprint. As demonstrated by UIC-CORBA, this approach offers interoperability with standards-based middleware. It does not directly support real-time assurances or time-bounded adaptation of middleware QoS properties, but as with MicroQoS CORBA, particular instantiations could address those challenges.

## 4 Motivation for nORB

### 4.1 Challenges for Middleware in the Boeing NEST OEP

There are two levels of middleware services in the Boeing OEP:

- infrastructure level middleware services
- domain level middleware services

The infrastructure middleware services includes the remote communication facilities and Dispatcher services provided by nORB. The domain level middleware services represent the deliverables for the NEST program.

This includes services like Group Management, Constraint Satisfaction, etc. which could be re-used across multiple NEST-class applications. In this paper, we exclusively deal with the infrastructure middleware services. Whenever we refer to *middleware*, we refer to infrastructure middleware services unless otherwise specified.

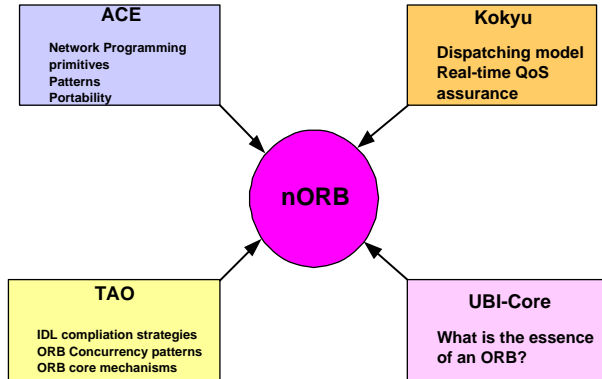


Figure 3: Re-use from existing frameworks

The key challenges for middleware design and implementation in the Boeing NEST OEP are to:

- provide a robust and portable DOC middleware framework,
- re-use existing infrastructure, patterns, and techniques,
- enforce real-time middleware QoS assurances,
- support time-bounded adaptation of middleware QoS properties,
- reduce middleware footprint, and
- facilitate accurate simulation of the OEP.

None of the frameworks described in Section 3 provides a complete answer to these challenges. ACE and Kokyu provides only primitive elements and TAO is a complete CORBA implementation which we did not need for the Boeing OEP because of footprint constraints. Above all, none of the ORB implementations provide an answer to the accurate simulation requirements described in more detail in Section 6. Our solution was to leverage the strengths of the frameworks which we found to be the most relevant to the Boeing OEP, as Figure 3 illustrates.

## 5 Architecture of nORB

In order to satisfy the communication needs of the Boeing OEP, we have developed a small footprint ORB called nORB. The NEST middleware services depend on nORB for providing remote communication services. nORB is built from primitive components from Adaptive Communication Environment [30]. It uses the CORBA [31] model of communication – the client side marshalling the parameters of a remote call into a request object and sending it to a remote server, which then demarshalls the request and calls the appropriate servant object. The reply is then marshalled into a reply object and sent back to the client, where it is demarshalled and returned to the caller. Although we are not compliant to the CORBA spec, we have re-used the main concepts in CORBA.

### 5.1 Message Formats

We use a limited subset of the type of messages supported by the CORBA specifications, so that we don't incur footprint bloat and at the same time support the bare minimum features required by the application. The following are the messages supported by nORB –

- Request message
- Reply message
- Locate Request message
- Locate Reply message

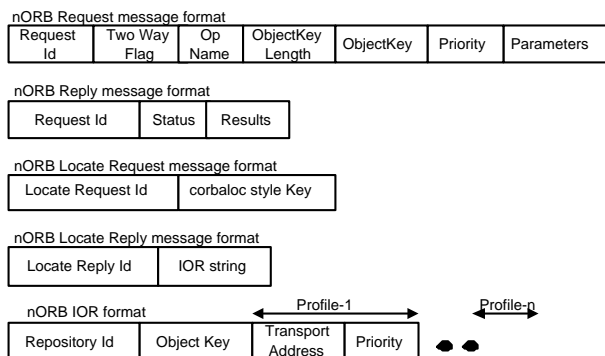


Figure 4: nORB IOR and Message formats

Figure 4 shows the formats of different messages in nORB. Notice that we only use a subset of the message

formats specified in the CORBA specification. The format of the Request and Reply messages closely resembles that of the GIOP Request and Reply messages respectively. We use Common Data Representation [31] to encode the messages themselves. The nORB client builds a Request message and sends it to the nORB server which sends a Reply back to the client. Based on the value of the *Two Way Flag* in the Request message, the client waits to get the Reply message from the server.

## 5.2 C++ Mapping of data types

We have developed a new C++ mapping for nORB which leverages the C++ Standard Template Library [32]. The mapping for simple types remain the same as in CORBA. For the more complex data types, the mapping shown in Table 1 is used.

IDL data type	C++ class
string	std::string
sequence	std::vector
Bounded Sequence	NORB::BoundedSequence
Array	NORB::Array

Table 1: nORB C++ Mapping

Since the CORBA mapping pre-dates the C++ standard, the original C++ mapping was not based on the new features available in C++. For example, an IDL sequence can be implemented using an STL vector. Array and BoundedSequence are implemented internally using vectors, but they differ from a sequence with respect to the fact that they provide bounds checking also.

## 5.3 IDL compiler

Interface Definition Language (IDL) offers a platform and language independent interface specification capability for ORB middleware. As in other areas, our solution seeks to adhere to the spirit of the CORBA standard, while departing from the standard as necessary to meet the stringent footprint and timeliness requirements of NEST-class systems. The design of the nORB IDL compiler is not only based on the TAO IDL compiler, but reuses some of its parts. We describe two main areas of work on IDL compilation for nORB. First, Section 5.3.1 discusses modifi-

cations to the TAO IDL compiler itself to remove capabilities not needed for NEST. Second, Section 5.3.2 describes changes to the servant skeletons generated by the IDL compiler to remove virtual functions.

### 5.3.1 TAO IDL Compiler Refactoring

The TAO IDL compiler is modular, consisting of a front-end (FE) library, a pluggable back-end (BE) library and a top-level executive. We were able to re-use the FE library as is, since its function is independent of code generation. We only made slight modifications to the top-level executive, to change the set of command line options. For the BE library, however, more significant changes were appropriate. First, stub and skeleton code generated from IDL was often tightly coupled to TAO internals to optimize performance and code reuse. Second, it was not so obvious initially to what extent the BE library would be reusable for nORB. However, previous refactoring of the TAO IDL compiler BE library, using design patterns such as Visitor and Factory Method, enabled a relatively straightforward retargeting of the BE library for the nORB IDL compiler, with significant re-use *of the portions of the BE library relevant to nORB*.

### 5.3.2 IDL Skeletons

IDL-generated skeleton classes are responsible for marshalling and demarshalling parameters and return values of interface methods and dispatching the remote calls to the appropriate servant object. In the standard CORBA C++ mapping, skeleton classes

- contain pure virtual functions for the IDL interface and marshalling/demarshalling functions, and
- use virtual inheritance to ensure correct behavior, and incidentally to reduce footprint, in cases where a C++ servant implements multiple IDL interfaces.

There are several problems with this approach, for NEST-class systems. First, virtual inheritance increases footprint in the single inheritance case, by requiring spurious storage of a virtual base class pointer in the object. Second, virtual functions may inhibit inlining the operation implementations. Third, the presence of virtual functions prevents safe in-memory access by multiple processes to

servants residing in shared memory, bypassing the transport layer. We now describe our optimizations to the IDL skeletons, to address these problems.

**Virtual Functions and Inheritance:** In many NEST-class systems, each object is likely to implement a single interface and thus inherit from a single skeleton base class. We therefore adopt the “curiously recurring template” pattern [33] to eliminate virtual inheritance and virtual functions in the skeletons. The skeleton classes become template classes that take the implementation classes as template parameters.

Although we remove virtual functions and virtual inheritance from the skeletons and thus the servants inheriting from them, one virtual function used by the transport layer for dispatching remote calls was unavoidable. Without virtual function polymorphism, the *nORB* transport layer could only dispatch upcalls to a single servant class, which would obviate the benefits of IDL. Fortunately, that virtual function is only accessed by the server process during a full remote invocation, and thus does not impact shared-memory access to the servant from different process address spaces.

**Avoiding Code Bloat:** One potential drawback to our approach is that it could result in code and footprint bloat, if there were more than one *nORB* object implementing the same interface on a given ORB. Our solution is to provide a mechanism allowing the application to choose whether to bind those interface operations statically or dynamically in the skeleton classes.

## 5.4 Object Adapter

In standard CORBA, each server-side ORB may provide multiple object adapters [34]. Servant objects register with an object adapter, which demultiplexes each client request to the appropriate servant. Each object adapter may be associated with a set of policies, *e.g.*, for threading, retention and lifespan [35]. In all compliant implementations of CORBA specifications, multiple object adapters are supported by each ORB. This allows heterogeneous object policies to be implemented in a client-server environment, which is desirable in applications such as on-line banking, where each object on a server may be configured according to preferences of the server administrator, or even the end user.

In *nORB*, however, there is no assumption of multiple object adapters. Instead, a single object adapter per ORB is considered preferable for simplicity and footprint reduction. In *nORB*, the number of objects hosted on a tiny node is expected to be very small, which reduces the need for multiple policies and thus for multiple object adapters.

Even though the resulting object adapter does not conform to the Portable Object Adapter specification, a significant amount of footprint reduction is achieved because of the reduced object adapter functionality.

We have simplified the process of object registration, to free developers from writing repetitive code as is seen in many CORBA programs. In the object adapter, we maintain a lookup table of object ids and pointers to servant implementation objects. The lookup table is synchronized using a Readers/Writer lock. We have also consolidated object registration with other setup functions, by moving it from the object adapter interface to the ORB interface.

## 5.5 Message Flow Architecture

In this section, we discuss the different strategies that are possible under different *norb* configurations to support the Request-Reply message flow model. Each configuration comes with its own advantages and disadvantages, which are discussed.

### 5.5.1 Reply wait strategy

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. The two-way function call semantics requires the caller thread to block until the reply comes back from the server. There are different strategies to wait for the reply, two of which we have chosen for implementation in *nORB*.

**Wait on connection** In this strategy, the caller thread makes the two-way call on the stub object, which then marshals the request and writes into the connection stream to the server using the appropriate socket handle. The client thread then calls *recv* on the socket expecting a reply from the server. The reply is read from the connection, the results are demarshalled and the caller thread

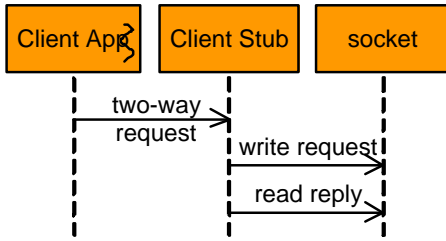


Figure 5: Waiting on connection for reply

proceeds. In this strategy, there is no need for a separate network thread to listen on the connection stream for a reply and hence there is no context switch overhead involved in processing the reply.

**Wait on condition variable** In this strategy, the sequence of calls is the same as the previous strategy until the request is written to the connection stream. After that, instead of waiting on the connection for the reply, the caller thread waits on a condition variable. A separate I/O thread is responsible for reading replies from the network connection. The network thread reads the reply header and payload, but it does not do the demarshalling, since the demarshalling will be done by the client stub in the context of the caller thread. The network I/O thread signals the condition variable, which unblocks the client thread. The client thread demarshals the reply and proceeds.

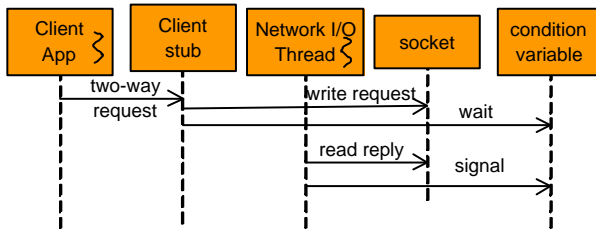


Figure 6: Waiting on condition variable for reply

In this approach, the reply processing is done in a thread different from the thread which makes the two-way function. Hence there is the overhead of synchronization and thread context-switch with this approach. If there is only one thread to process the reply and if many threads make two-way function calls simultaneously, then the *Request id* in the request and reply header can be used to correlate a received reply with a sent request. A singleton *Request id* generator is used to generate the *Request id* to be sent with the next request. Since there are multiple

threads waiting for replies, the replies can be inserted into a *set* and the condition variable signalled. Each thread wakes up and checks whether the reply, that it has been waiting for, has arrived. The reply is correlated to the request using the *Request id* field in the reply. If not, the thread blocks again on the condition variable.

### 5.5.2 Upcall Dispatch Strategy

On the server side there are different strategies to process an incoming request and send the reply back to the client. We discuss two different strategies which we have implemented in nORB.

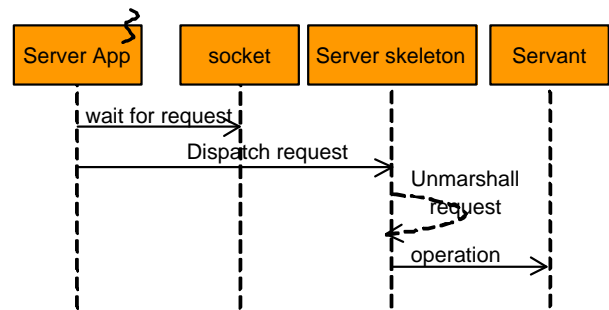


Figure 7: Direct upcall strategy

**Direct Upcall Strategy** In this strategy, the upcall is dispatched in the same thread as the I/O thread which listens for incoming requests from the connection stream. Figure 7 shows the sequence of events that take place. The server thread waits on the *select* call waiting for incoming data on the network connection. As soon as a complete request is read, the object key is obtained from the request. The Object Adapter is queried using the object key to obtain the skeleton object for which the operation is destined. The skeleton object takes the responsibility of demarshalling the parameters in the request and then calling the appropriate method on the C++ object. Since there is only thread, there is no need for any context switches or synchronization overhead. The disadvantage is that the network thread is blocked until the upcall is done and hence all the other incoming requests will be queued in the TCP buffers until the upcall is done and the network I/O thread goes back to the listening mode. It is possible to overcome this limitation by using the Leader-Followers pattern [36, 5], where multiple threads take turns listening to the network and making the upcall.

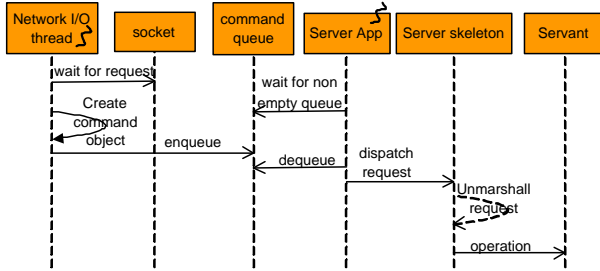


Figure 8: Queued upcall strategy

**Queued Upcall Strategy** This strategy follows the Half Sync Half Async pattern [37, 5]. In contrast with the direct upcall strategy, there is a network I/O thread dedicated to receiving requests from clients. Once the request is received, it is encapsulated into a command object and then enqueued into a queue. This queue would be shared with another thread, in which the upcall dispatch is done. The dispatch thread dequeues command objects from the queue, extracts the request, finds the servant object using the object key and dispatches the operation. The advantage of this approach is that the network processing and the actual operation dispatching are decoupled. But there is the overhead of an extra context-switch and queue synchronization with this approach.

## 5.6 Dispatcher

As shown in Figure 1, control events are triggered at various rates in the Boeing OEP. The Dispatcher is used to trigger these events. The nORB Dispatcher is heavily based on the Kokyu [8] dispatching model. Kokyu [7] is a low-level middleware framework built on ACE, for flexible multi-paradigm scheduling [8] and configurable dispatching of real-time operations. Kokyu abstracts combinations of fundamental real-time scheduling and dispatching elements to enforce a variety of real-time policies, including well-known strategies such as Rate-Monotonic Scheduling (RMS) [9], Earliest Deadline First (EDF) [9], and Maximum Urgency First [10].

The Kokyu Dispatcher framework provides the infrastructure for triggering events on the client side and for dispatching the upcalls on the server side which includes the threads, timers and queues on both sides. Reordering of requests can be done on the server side if necessary. The Dispatcher is used in association with a scheduler which sets up the appropriate number of lanes and their prior-

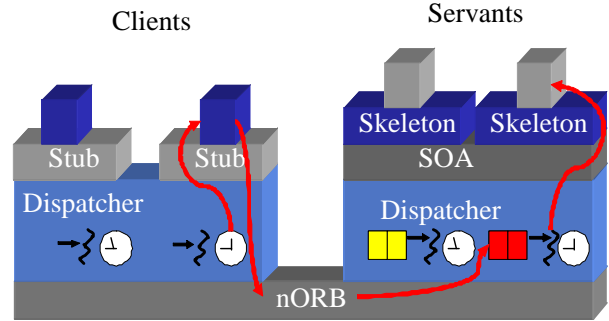


Figure 9: Kokyu dispatcher in nORB

ities based on a scheduling strategy. In the case of the Boeing OEP, the scheduler is a reusable middleware service being developed by NEST researchers. A *Dispatchable* interface is provided, which is to be implemented by an application object that needs to be time-triggered. The *handle\_dispatch()* method will be called on the *Dispatchable*. In the Boeing OEP, some application objects are *Dispatchables* and make remote calls when triggered. Figure 9 shows the message flow from the client to server when a timer goes off on the client side.

## 5.7 Real Time assurance

nORB borrows the concepts in TAO to implement real-time priority propagation [38, 39]. The client ORB uses the priority of the thread in which the remote invocation is made. nORB then looks for a matching priority from the set of profiles in the IOR and then makes a connection to the appropriate port. We use a cached connection strategy [40] to avoid the overhead of a connection setup everytime a remote invocation is made. To alleviate priority inversion, each connection endpoint on the server is associated to a thread/reactor pair. The priority of the thread associated with a dispatching lane is set appropriately so that a request coming in to a higher priority lane will be processed before a request coming in to a lower priority lane. Figure 10 shows an example where Rate Monotonic Scheduling [9] is used to assign priorities to the different rates on the client side.

The priority of the client thread making the request is propagated in the request. This priority is used on the server side to enqueue the request if necessary as explained in Section 6.

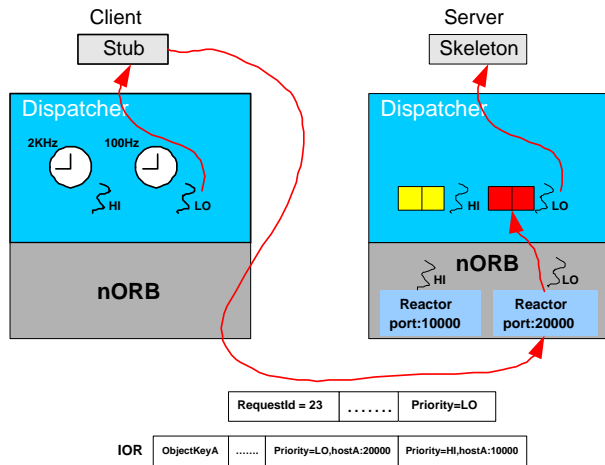


Figure 10: Priority Propagation in nORB

## 6 Clock Synchronization

### 6.1 Motivation

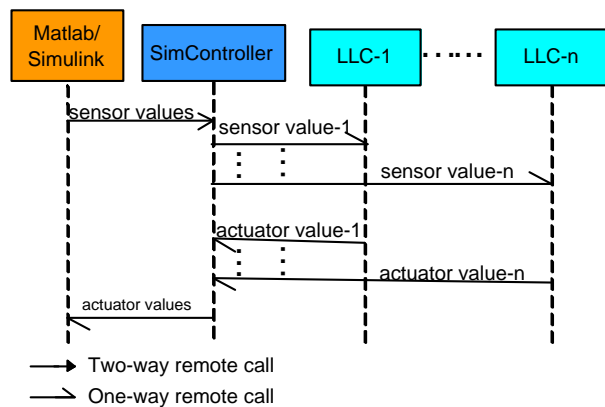


Figure 11: Sequence diagram for Matlab-OEP interaction

**OEP Simulation** The Boeing NEST OEP uses Matlab to simulate the sensor-actuator network hardware. The *SimController* acts as a multiplexer/demultiplexer for the data which flows between Matlab and the nodes similar to the Gather-write Scatter-Read [41] pattern. Figure 11 illustrates the sequence of events which take place during each data exchange between Matlab and *SimController*. Matlab makes a two way call passing the sensor values for all the nodes and it blocks expecting the actuator values from all the nodes. The sensor values from Matlab are collected and then distributed to the individual nodes by the

*SimController* using remote one-way calls. Each node, after getting the sensor value, does the necessary control processing and sends the actuator value to the *SimController* using a one-way call. The actuator values from all the nodes are collected by the *SimController* and passed to Matlab. This completes one low-level controller cycle, which runs at 2KHz.

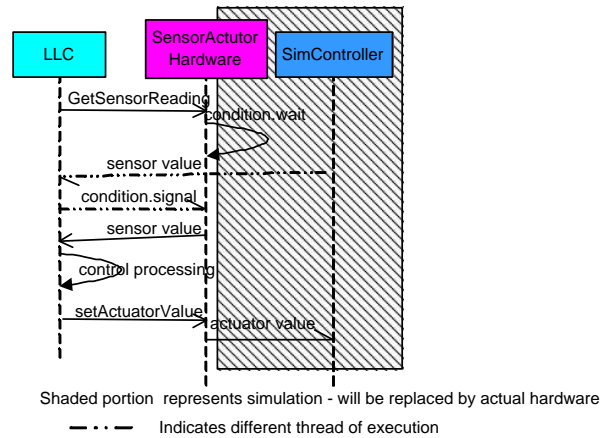


Figure 12: Sequence diagram for LLC-hardware interaction

Figure 12 illustrates the sequence of interactions between the low level controller nodes and *SimController*. With the actual hardware in place, the *GetSensorReading* call will return immediately with the sensor reading from the sensor. In contrast, in simulation, this call will block until the Matlab simulator sends the sensor value for that node. *SimController* acts as an adapter [42] translating Matlab calls into nORB based remote calls and vice-versa. The *GetSensorReading* call itself is a local function call, even though its implementation will generate a remote one-way call while in simulation mode. Once the remote one-way call is made, the LLC will block on a condition variable waiting for sensor values from *SimController*. This is a variant of the Half-sync/Half-async pattern [5, 37] with a queue size of 1. The synchronous *GetSensorReading* call is converted into an asynchronous wait.

**Need for a Logical Clock** Since a whole lot of remote communication is involved in simulation, it is not surprising that one LLC cycle (2KHz) is going to take more than 0.0005 seconds. As explained in Section 2, a particular node could be configured to perform any logical functionality at any given point in time. If all nodes were per-

forming the function of an LLC, then the *SimController* would implicitly provide a kind of barrier synchronization among all the nodes. But this is not true and hence nodes which are not performing the LLC function will drift out of sync with respect to the ones which do. This is especially true, if the nodes are distributed across different hosts in the simulation run. The following are the challenges involved in simulation of the OEP:

- We need to be able to use the same software in simulation as in the final OEP. This helps us to obtain relatively faithful metrics as opposed to obtaining *some* metrics using the classical distributed event simulation.
- We could have arbitrary deployment configurations for the simulation. For example, for a 100 node simulation, we could run the 12 group controller nodes in one machine and other 88 nodes on a different machine and run Matlab and *SimController* on yet another machine. The hardware and software configuration may be different for each machine and hence the timer expiry intervals may not be the same across the different nodes.
- The idea of time scaling will not work since it does not guarantee that the nodes are in sync. Time has to be scaled in such a manner that all the computation and communication times are known. This would be highly error prone. Moreover, even if we scale the time to a *safe* upper bound, the time (wall-clock time) it actually takes to run the simulation would be prohibitively large. Again, as explained before, this won't work with a heterogeneous environment.
- Because of the heterogeneous configuration some nodes might run faster than others and hence the slower running nodes would get *messages from the future* from the faster running nodes. We want to avoid processing the future messages on a slower nodes until the node "catches up".
- Some infrastructure is necessary which will hide the heterogeneity of the different environments and simulate the dispatcher of a real-time OS on top of a non real-time OS.

## 6.2 Solution

Our solution is to have nORB maintain a logical clock at each node. Even though a node is not configured to function as a LLC, it is forced to function as a LLC, so as to synchronize with the rest of the nodes. There won't be any actual processing of the sensor values passed to such dummy LLC nodes and the actuator values passed back to the simulator will be zeroed out. Time is divided into discrete frames, each frame interval representing an LLC cycle interval (0.5 msec). Note that the frame interval is in logical clock units and most probably would be different from the real time interval of 0.5 msec for an LLC cycle. The logical clock would be incremented in discrete units by nORB after any incoming message is processed. If there are no incoming messages to be processed, then the time is incremented to the start of the next frame.

Since each node acts like an LLC, at the start of each frame, the LLC is triggered which results in the node waiting for sensor values from Matlab. While a node is waiting for the sensor values, we "freeze" the logical clock on that node, i.e. we prevent the logical clock from advancing. As a result, no messages are processed on that node and all the incoming messages are queued. The queuing of incoming messages enables the messages from the future to be processed when a slower node "catches up".

A request or reply leaving a node would carry the current logical clock value on that node. This value will be used by the receiving node to sort the incoming messages based on their time of release and the logical clock of the receiving node. If the logical clock of the receiving node is higher than that of the incoming message, then the message is stamped with the logical clock at the receiving node.

Figure 13 shows the sequence of events that take place with the addition of the logical clock thread.

1. When the logical clock advances, the clock thread goes through the list of dispatchables to see whether there are any dispatchables eligible to be triggered. An eligible dispatchable is one whose next trigger time is less than or equal to the current logical clock and the previous execution of this dispatchable has completed execution.
2. If there are any eligible dispatchables, the dispatchable is released to the clock thread's queue. Based

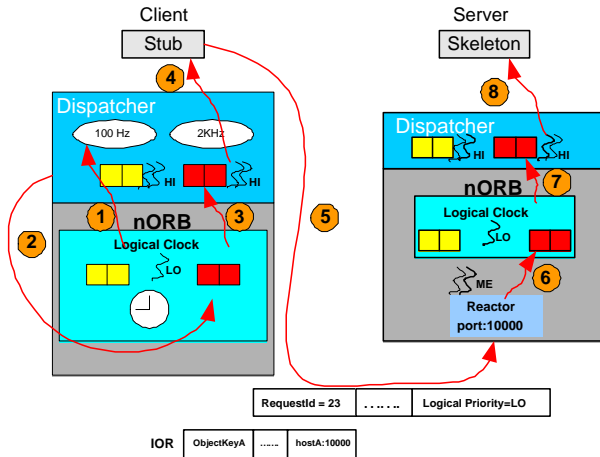


Figure 13: Message processing with logical clock

on the priority assigned to each dispatchable, it is enqueued into the appropriate queue.

3. The clock thread selects the earliest item (message or dispatchable) from among the different priority queues. Note that each queue contains the incoming request and reply messages also, which will then compete for upcall dispatching with the locally released dispatchables. The clock thread then enqueues the selected item in the appropriate priority lane of the Dispatcher.
4. The lane thread in the Dispatcher dispatches the enqueued item. This might inturn involve a remote call to a servant object.
5. The priority of the dispatchable or the message is propagated to the server side. Currently the application scheduler uses RMS to decide the priority of the dispatchable based on their rates. Each lane thread stores its assigned priority in Thread Specific Storage [5]. The native priorities of all the lane threads are the same while we using the clock simulation.
6. The incoming message is accepted by the reactor thread and enqueued in the clock thread of the server. Notice that there is only one reactor thread which runs at a medium priority level when compared to the clock and the lane threads. This is different from the previous approach discussed in Section 5.7 and is applied only when using the clock simulation. The

lane threads are given the maximum priority and the clock thread itself is assigned the least priority. We found that this setup is optimal for the simulation run while ensuring synchronization between the different nodes in the simulation. If the clock thread were given a higher priority, the initialization process for the OEP takes an unusually long time. Moreover, we would like minimum delay processing the incoming request/reply messages.

7. As on the client side, the clock thread chooses the most eligible item from the queue and enqueues it on the appropriate lane thread's queue.
8. Lane thread dispatches the enqueued item.

The following algorithm will be used by a single clock thread to process each incoming message. Along with the messages, the time-triggered dispatchables are also queued. An *item* in the algorithm represents a message or a dispatchable.

```

loop forever
{
  get the next earliest item to be processed.
  The queues are traversed in priority order
  to find the earliest item.

  if no item found
  {
    if dispatchables to be triggered
    {
      increment logical clock to next dispatchable
      release time.
      Notify dispatcher so that the appropriate
      dispatchables are released in to the queues,
      if needed.
    }
  }
  else
  {
    Queue is empty. Wait till queue not empty.
  }
}
else item found and item.release_time
  > current_clock
{
  This is a message from the future.
  We cannot process it right now.
  increment clock to
  min(arrival time of earliest item on queue,
  next frame start time)

  Notify dispatcher so that the appropriate
  dispatchables are released, if needed.
}
else if item found and
  item.release_time <= current_clock
{
  This is a message from the past.
}

```

```

We can process it right now.
available_time is the time available till
the next frame start time
or next_earliest_arrival_time, which ever
is earlier. next_earliest_arrival_time will
be the next release time for a higher
priority dispatchable or an incoming message.

next_earliest =
    min(next higher priority dispatchable/
        message arrival time,
        next frame start time)

available_time = next_earliest -
                item.execution_time

if available_time >= item.execution_time
{
    //item will finish execution before the next
    //release time.
    item.execute()
    //this will result in an upcall or
    //dispatchable.handle_dispatch()

    curr_clock += item.execution_time
    Notify dispatcher so that the appropriate
    dispatchables are released, if needed.
}
else if available_time < item.execution_time
{
    //item uses up whatever available time
    item.execution_time -= available_time
    curr_clock += available_time
    Enqueue item again into appropriate queue.
    Notify dispatcher so that the appropriate
    dispatchables are released, if needed.
}
}

```

Each *item* carries an execution time associated with it, which is predefined. When an *item* is eligible for execution, the clock thread dequeues it and checks whether it can complete its execution before the next dispatchable release time or the next eligible request/reply message from the future. If it can complete its execution, the clock thread enqueues it in the appropriate *lane*. If not, it simulates the execution slice of the item and keeps track of the remaining execution time in the item itself and the item is then enqueued back to the clock thread's queue itself so that it can compete with other items for execution eligibility. A *lane* could be running a single thread or a pool of worker threads. As described in Section ??, each lane serves messages of a particular priority. But since the time has already been accounted for by the logical clock thread, all the lane threads across different lanes run at the same priority. The logical priority of each lane is still maintained in thread specific storage [5] in each of the lanes, so that the logical priority can be sent with the re-

quest messages.

## References

- [1] DARPA IXO, "Networked Embedded Software Technology (NEST)." <http://www.darpa.mil/ixo/>.
- [2] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture—A System of Patterns*. New York: Wiley & Sons, 1996.
- [3] D. C. Schmidt, "ACE: an Object-Oriented Framework for Developing Distributed Applications," in *Proceedings of the 6<sup>th</sup> USENIX C++ Technical Conference*, (Cambridge, Massachusetts), USENIX Association, Apr. 1994.
- [4] Center for Distributed Object Computing, "The ADAPTIVE Communication Environment (ACE)." [www.cs.wustl.edu/~schmidt/ACE.html](http://www.cs.wustl.edu/~schmidt/ACE.html), Washington University.
- [5] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, Volume 2*. New York: Wiley & Sons, 2000.
- [6] D. C. Schmidt, "An Architectural Overview of the ACE Framework: A Case-study of Successful Cross-platform Systems Software Reuse," *login.*, Nov. 1998.
- [7] C. D. Gill, R. Cytron, and D. C. Schmidt, "Middleware Scheduling Optimization Techniques for Distributed Real-Time and Embedded Systems," in *Proceedings of the 7<sup>th</sup> Workshop on Object-oriented Real-time Dependable Systems*, (San Diego, CA), IEEE, Jan. 2002.
- [8] C. Gill, D. C. Schmidt, and R. Cytron, "Multi-Paradigm Scheduling for Distributed Real-Time Embedded Computing," *IEEE Proceedings Special Issue on Modeling and Design of Embedded Software*, Oct. 2002.
- [9] C. Liu and J. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," *JACM*, vol. 20, pp. 46–61, Jan. 1973.
- [10] D. B. Stewart and P. K. Khosla, "Real-Time Scheduling of Sensor-Based Control Systems," in *Real-Time Programming* (W. Halang and K. Ramamritham, eds.), Tarrytown, NY: Pergamon Press, 1992.
- [11] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, 2.4 ed., Oct. 2000.
- [12] SUN, "Java Remote Method Invocation (RMI) Specification." [java.sun.com/products/jdk/1.2/docs/guide/rmi/spec/rmi-TOC.doc.html](http://java.sun.com/products/jdk/1.2/docs/guide/rmi/spec/rmi-TOC.doc.html), 2002.
- [13] Object Management Group, *Real-time CORBA Joint Revised Submission*, OMG Document orbos/99-02-12 ed., Mar. 1999.
- [14] Object Management Group, *Dynamic Scheduling Real-Time CORBA Joint Revised Submission*, OMG Document orbos/2000-08-12 ed., Aug. 2000.
- [15] Bollella, Gosling, Brosgol, Dibble, Furr, Hardin, and Turnbull, *The Real-Time Specification for Java*. Addison-Wesley, 2000.
- [16] E. D. Jensen, "Distributed Real-Time Specification for Java." [java.sun.com/aboutJava/communityprocess/jst/jsr\\_050\\_drt.html](http://java.sun.com/aboutJava/communityprocess/jst/jsr_050_drt.html), 2000.

- [17] D. C. S. et. al, "TAO: A Pattern-Oriented Object Request Broker for Distributed Real-time and Embedded Systems," *IEEE Distributed Systems Online*, vol. 3, Feb. 2002.
- [18] Center for Distributed Object Computing, "The ACE ORB (TAO)." [www.cs.wustl.edu/~schmidt/TAO.html](http://www.cs.wustl.edu/~schmidt/TAO.html), Washington University.
- [19] C. O’Ryan, D. C. Schmidt, F. Kuhns, M. Spivak, J. Parsons, I. Pyarali, and D. Levine, "Evaluating Policies and Mechanisms for Supporting Embedded, Real-Time Applications with CORBA 3.0," in *Proceedings of the 6<sup>th</sup> IEEE Real-Time Technology and Applications Symposium*, (Washington DC), IEEE, May 2000.
- [20] I. Pyarali, D. C. Schmidt, and R. Cytron, "Achieving End-to-End Predictability of the TAO Real-time CORBA ORB," in *8<sup>th</sup> IEEE Real-Time Technology and Applications Symposium*, (San Jose), IEEE, Sept. 2002.
- [21] 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.
- [22] Object Management Group, *Minimum CORBA - Joint Revised Submission*, OMG Document orbos/98-08-04 ed., Aug. 1998.
- [23] David McKinnon, et al., "MicroQoS CORBA." <http://microqoscorba.eecs.wsu.edu/>.
- [24] A. D. McKinnon and D. Bakken and J. Shovic, "Micro-QoS CORBA: A Reflective, QoS-Enabled, Configurable MicroCORBA With CASE Support," in *Proceedings of the Second Workshop on Real-time and Embedded Distributed Object Computing*, OMG, June 2001.
- [25] A. D. McKinnon and O. Haugan and T. Damania and D. Bakken and J. Shovic, "MicroQoS CORBA: A QoS-Enabled, Reflective, and Configurable Middleware Framework for Embedded Systems." <http://microqoscorba.eecs.wsu.edu/MicroQoS CORBA-November2001.pdf>.
- [26] M. Roman, M. D. Mickunas, F. Kon, and R. H. Campbell, "LegORB and Ubiquitous CORBA," in *Reflective Middleware Workshop*, ACM/IFIP, Apr. 2000.
- [27] Manuel Roman and Roy H. Campbell and Fabio Kon, "Reflective Middleware: From Your Desk to Your Hand," *IEEE Distributed Systems Online*, vol. 2, July 2001.
- [28] Manuel Roman, "UbiCore: Universally Interoperable Core." [www.ubi-core.com/Documentation/Universally\\_Interoperable\\_Core/universally\\_interoperable\\_core.html](http://www.ubi-core.com/Documentation/Universally_Interoperable_Core/universally_interoperable_core.html).
- [29] J. Snell and K. MacLeod, *Programming Web Applications with SOAP*. O’Reilly, 2001.
- [30] D. C. Schmidt and S. D. Huston, *C++ Network Programming, Volume 2: Systematic Reuse with ACE and Frameworks*. Reading, Massachusetts: Addison-Wesley, 2002.
- [31] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, 3.0 ed., June 2002.
- [32] A. Stepanov and M. Lee, "The Standard Template Library," Tech. Rep. HPL-94-34, Hewlett-Packard Laboratories, Apr. 1994.
- [33] J. Coplien, "Curiously recurring template patterns," *C++ Report*, vol. 7, pp. 40–43, Feb. 1995.
- [34] I. Pyarali and D. C. Schmidt, "An Overview of the CORBA Portable Object Adapter," *ACM StandardView*, vol. 6, Mar. 1998.
- [35] M. Henning and S. Vinoski, *Advanced CORBA Programming with C++*. Reading, MA: Addison-Wesley, 1999.
- [36] D. C. Schmidt, C. O’Ryan, I. Pyarali, M. Kircher, and F. Buschmann, "Leader/Followers: A Design Pattern for Efficient Multi-threaded Event Demultiplexing and Dispatching," in *Proceedings of the 6th Pattern Languages of Programming Conference*, (Monticello, IL), Aug. 2000.
- [37] D. C. Schmidt and C. D. Cranor, "Half-Sync/Half-Async: an Architectural Pattern for Efficient and Well-structured Concurrent I/O," in *Proceedings of the 2<sup>nd</sup> Annual Conference on the Pattern Languages of Programs*, (Monticello, Illinois), pp. 1–10, Sept. 1995.
- [38] D. C. Schmidt, S. Mungee, S. Flores-Gaitan, and A. Gokhale, "Alleviating Priority Inversion and Non-determinism in Real-time CORBA ORB Core Architectures," in *Proceedings of the 4<sup>th</sup> IEEE Real-Time Technology and Applications Symposium*, (Denver, CO), IEEE, June 1998.
- [39] D. C. Schmidt, S. Mungee, S. Flores-Gaitan, and A. Gokhale, "Software Architectures for Reducing Priority Inversion and Non-determinism in Real-time Object Request Brokers," *Journal of Real-time Systems, special issue on Real-time Computing in the Age of the Web and the Internet*, vol. 21, no. 2, 2001.
- [40] D. C. Schmidt and C. Cleeland, "Applying a Pattern Language to Develop Extensible ORB Middleware," in *Design Patterns in Communications* (L. Rising, ed.), Cambridge University Press, 2000.
- [41] W. R. Stevens, *UNIX Network Programming, Volume 1: Networking APIs: Sockets and XTI, 2nd Edition*. Englewood Cliffs, NJ: Prentice Hall, 1998.
- [42] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.