A Pattern Language for Efficient, Predictable, Scalable, and Flexible Dispatching Mechanisms for Distributed Object Computing Middleware

Irfan Pyarali and Carlos O'Ryan
{irfan,coryan}@cs.wustl.edu
Department of Computer Science, Washington University
St. Louis, MO 63130, USA

Douglas C. Schmidt
schmidt@uci.edu
Electrical and Computer Engineering Dept.
University of California, Irvine, CA 92697*

This paper has been submitted to the IEEE/IFIP International Symposium on Object-Oriented Real-time Distributed Computing, March 15-17, 2000, Newport Beach, California.

Abstract

In many application domains, the distributed object computing (DOC) middleware is responsible for dispatching upcalls to one or more application objects when events or requests arrive from clients. Implementing efficient, predictable, and scalable middleware dispatching mechanisms is hard and implementing them for multi-threaded systems is even harder. In particular, dispatching mechanisms must be prepared to dispatch upcalls to multiple objects, to handle recursive requests originated from application-provided upcalls, and must often collaborate with the application to control object' life-cycle.

In our DOC middleware research we have implemented many dispatching mechanisms that repeatedly apply common solutions to solve the challenges outlined above. Moreover, we have discovered that the forces constraining dispatching mechanisms often differ slightly, thereby requiring alternative solutions. This paper presents two contributions to the design and implementation of efficient, predictable, scalable, and flexible DOC middleware and applications. First, it shows how patterns can be applied to the object-oriented systems to capture key design and performance characteristics of proven dispatching mechanisms. Second, it presents a pattern language that describes successful solutions that are appropriate for key dispatching challenges that arise in various real-time DOC middleware and applications.

Keywords: Frameworks; Design Patterns; Real-Time Distributed Object Computing

*This work was funded in part by Boeing, NSF grant NCR-9628218, DARPA contract 9701516, and Siemens.

1 Introduction

Dispatching mechanisms form the core of distributed object computing (DOC) middleware. For instance, the dispatching mechanisms in a CORBA Object Request Broker (ORB) are responsible for dispatching incoming client *events* or *requests* to other (1) ORB components and (2) the application-level objects that implement application-defined behavior. In general, DOC middleware dispatching mechanisms must handle a variety of tasks, such as (1) dispatching multiple requests simultaneously, (2) handling recursive dispatches from within application-provided upcalls, (3) efficiently dispatching the same upcall to multiple objects, and (4) adding and removing objects in the dispatching table while upcalls are in progress.

This paper presents a family of related patterns, known as a *pattern language* [1], that we have used to develop efficient, predictable, and scalable DOC middleware dispatching mechanisms in a variety of application domains, including the TAO Real-Time CORBA [2] ORB [3], real-time avionics mission computing with strict periodic dead-line requirements [4], and distributed interactive simulations with high scalability requirements [5]. In addition, the patterns in our pattern language have been used to implement various dispatching-oriented framework components, such as Reactors [6], Proactors [7], Observers [8], and Model-View-Controllers [9].

The remainder of this paper is organized as follows: Section 2 describes the context for dispatching mechanisms and identifies common requirements for several typical use-cases; Section 3 presents the pattern language used to implement efficient, predictable, scalable, and flexible dispatching mechanisms for both single and multiple targets; and Section 4 presents concluding remarks.

2 An Overview of Dispatching Mechanisms and Patterns

This section summarizes the functionality and requirements of two common use-cases that illustrate the challenges associated with developing dispatching mechanisms for DOC middleware. The first example is the Object Adapter [10] component in a standard CORBA [11] ORB. The second example is a Event Channel in the standard CORBA Event Service [12].

Object Adapter dispatching mechanisms: The core responsibilities of a CORBA Object Adapter include (1) generating identifiers for objects that are exported to clients and (2) mapping subsequent client requests to the appropriate object implementations, which are called *servants* in the CORBA model. Figure 1 illustrates the general structure and interactions of a CORBA Object Adapter.

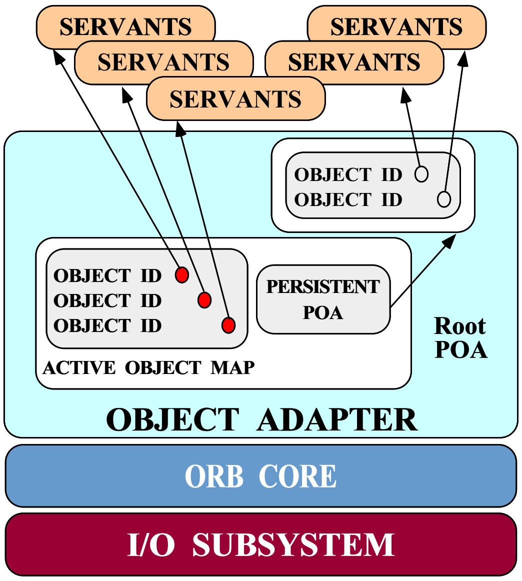

Figure 1: Object Adapter Structure and Interactions

In addition to its core responsibilities, a CORBA Object Adapter must correctly deal with the following situation:

• **Non-existent objects:** Clients may invoke requests on "stale" identifiers, *i.e.*, for objects that have been deactivated from the Object Adapter. In this case, the Object Adapter should not use the stale object pointer because it may have been deleted by the application. Instead, it must propagate the appropriate exception back to the client.

• **Unusual object activation/deactivation use-cases:** Object Adapters are responsible for activating and deactivating objects on-demand. Moreover, server application objects can activate or deactivate other objects in response to a client request. An object can even deactivate itself while in its own upcall, *e.g.*, if the request is a "destroy yourself" message.

• **Multi-threading hazards:** Implementing an Object Adapter that works correctly and efficiently in a multi-threaded environment is hard. For instance, there are many opportunities for deadlock, reduced concurrency, and priority inversion that may arise from recursive calls to an Object Adapter while it is dispatching requests. Likewise, excessive synchronization overhead may arise from locking performed on the dispatching table.

Event Channel dispatching mechanisms: The CORBA Event Service defines *supplier* and *consumer* participants that are designed to alleviate some restrictions [4] with standard CORBA invocation models. As shown in Figure 2 suppliers

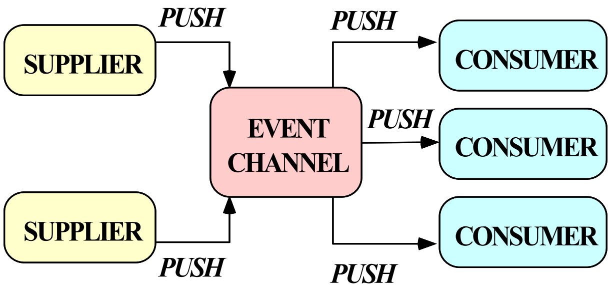

Figure 2: Participants in the COS Event Service Architecture

generate events and consumers process events received from suppliers. This figure also illustrates the *Event Channel*, which is a mediator [8] that dispatches events to consumers on behalf of suppliers. By using an Event Channel, events can be delivered from suppliers to consumers without requiring the participants to know about each other explicitly.

To perform its core responsibilities, a CORBA Event Channel must address the following aspects:

• **Dynamic consumer subscriptions:** A robust implementation of an Event Channel must cope with new consumers being added while dispatching is in progress. Likewise, it must cope with existing consumers removing themselves before all active dispatching operations complete. In multi-threaded environments, it is possible for threads, potentially running at different priorities, to iterate over the dispatching table concurrently. Some consumers may trigger further updates, which also must be handled properly and efficiently.

Näive implementations, such as copying the complete set of consumers before starting the iteration, may fail if one consumer is destroyed as a side-effect of the upcall on another consumer. In multi-threaded implementations, the problem is exacerbated because separate threads may remove and destroy elements in the table simultaneously.

• **Variable dispatching times:** Dispatching events in an Event Channel requires iteration over the set of consumers. The use of an iterator makes it even harder to provide predictable implementations since the number of consumers may vary. Thus, some type of synchronization is required during the dispatching process.

Implementations of the Observer pattern [8] must also contend with problems similar to CORBA Event Service. The Observer pattern propagates updates to a component emanating from one or more suppliers to multiple consumers, *i.e.*, observers. An implementation of this pattern must iterate over the set of consumers and communicate the update to each one of them. As with the Event Channel, subscriptions may change dynamically while updates are being dispatched.

Historically, a variety of *ad hoc* techniques have emerged to address the dispatching challenges outlined above. No one technique is optimal for all application domains, however. For instance, real-time implementations may impose too much overhead for high-performance, best-effort systems. Likewise, implementations tailored for multi-threading may pose excessive locking overhead for single-threaded reactive systems. In addition, solutions that deal with recursive access can incur excessive overhead if all upcalls are dispatched to separate threads or remote servers. Thus, what is required are a techniques of techniques and methodologies that systematically capture the range of possible solutions that arise in the design space of dispatching mechanisms. One such approach is described in the following section.

3 A Pattern Language for Dispatching Mechanisms

Certain patterns, such as Strategized Locking [13] or Strategy [8] address some of the challenges associated with developing efficient, predictable, scalable, and flexible dispatching mechanisms. In other cases, however, the relationships and collaborations between components in dispatching mechanisms require more specialized solutions. Moreover, as noted in Section 2, no single pattern or solution resolves all the forces faced by developers of complex dispatching mechanisms. Therefore, this section presents a *pattern language* that addresses the challenges for dispatching mechanisms outlined in Section 2.

A pattern is a recurring solution to a standard problem within a particular context [8]. When related patterns are woven together they form a "language" [1] that provides a process for the orderly resolution of software development problems. Pattern languages are not formal languages or programming languages, but rather a collection of interrelated patterns that provide a vocabulary for solving particular problems [9]. Both patterns and pattern languages help developers communicate architectural knowledge, help developers learn a new design paradigm or architectural style, and help new developers avoid traps and pitfalls that have traditionally been learned only by costly experience [14].

Each pattern in our dispatching mechanism pattern language resolves a particular set of forces, with varying consequences on performance, functionality, and flexibility. In general, simpler solutions result in better performance, but do not resolve all the forces that more complex dispatching mechanisms can handle. However, application developers should not disregard simpler patterns. Instead, they should apply the patterns that are most appropriate for the problem at hand, balancing the need to support advanced features with the performance and flexibility requirements of their applications.

3.1 Dispatching to a Single Object

The patterns described in this section focus on middleware mechanisms where events or requests are dispatched to a single target object. The initial patterns are somewhat simple and are geared toward less complicated systems. The latter patterns are more intricate and can satisfy more complex requirements for efficiency, predictability, scalability, and flexibility.

3.1.1 Serialized Dispatching

Context: Dispatching mechanisms are vital components in DOC middleware and applications. They typically contain a collection of target objects that reside in a dispatching table. This table is used to select the appropriate object based upon an identifier contained in an incoming request. For example, the CORBA architecture [11] defines an Object Adapter [10] that (1) maps client requests to objects supplied by server applications and (2) helps dispatch operations on the server objects.

Problem: Multi-threaded applications must serialize access to their dispatching table to prevent data corruption.

Forces: Serialization mechanisms, such as mutexes, are relatively expensive and can incur priority inversion and non-determinism. Thus, their use should be minimized for high-performance and real-time systems. In addition, applications should be able to safely add and remove objects registered with the dispatching table during run-time.

Solution: Use the Monitor Object pattern [15] where a single monitor lock serializes access to the entire dispatching table, as shown in Figure 3. The monitor lock is held both while (1) searching the table to locate the object and (2) dispatching the appropriate operation call on the application-provided code. In addition, the same monitor lock is used when inserting and removing entries from the table.

Consequences: A regular monitor lock is sufficient to achieve the serialization necessary for this dispatching mechanism. Since only one set of acquire/release calls are made on the lock during an upcall, the serialization overhead is minimal. Thus, this design is appropriate when there is little or no

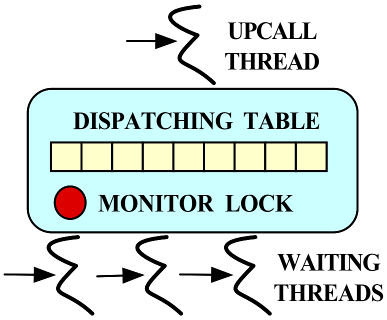

Figure 3: Serialized Dispatching with a Monitor Lock

contention for the dispatching table or when upcalls to application code are short-lived.

A simple protocol can control the life-cycle of objects registered with the dispatching mechanism. For instance, an object cannot be destroyed while it is still registered in the dispatching table. Since the table's monitor lock is used both for dispatching and modifying the table, other threads cannot delete an object that is in the midst of being dispatched.

Note, however, that this pattern may be inadequate for systems with stringent real-time requirements. In particular, the monitor lock is held during the execution of application code, making it hard for the dispatching mechanism to predict how long it will take to release the monitor lock. Likewise, this pattern does not work well when there is significant contention for the dispatching table, *e.g.*, if two requests arrive simultaneously for different target objects in the same dispatching table, only one of them can be dispatched at a time.

3.1.2 Serialized Dispatching with a Recursive Mutex

Context: Implementing the dispatching mechanism in multi-threaded DOC middleware, as outlined in Section 3.1.1.

Problem: Monitor locks on many OS platforms are not recursive. Thus, once the dispatching table's lock has been acquired, any attempt to query or modify the dispatching table as part of an application upcall will result in a deadlock.

Forces: A monitor lock cannot be released before dispatching the application upcall because another thread could remove and destroy the object while it is still being dispatched.

Solution: Use a *recursive* monitor lock [16], which allows the calling thread to re-acquire it if that thread already owns the monitor lock. The structure of this solution is identical to the one shown in Figure 3, except that a recursive monitor lock is used instead of a non-recursive lock.

Consequences: As before, the monitor lock serializes concurrent access to avoid corruption of the dispatching table. However, unlike the Serialized Dispatching pattern outlined in Section 3.1.1, application upcalls can modify the dispatching table or dispatch new upcalls.

Unfortunately, this solution does not adequately resolve the concurrency and predictability problems since the monitor is held through the upcall. Thus, it is still hard for the dispatching mechanism to predict how long it will take to release the monitor lock and the mechanism does not allow multiple requests to be dispatched simultaneously. Moreover, recursive monitor locks are usually more expensive than their non-recursive counterparts [17].

3.1.3 Dispatching with a Readers/Writer Lock

Context: In complex DOC middleware and applications, events and requests often occur simultaneously. Unless application upcalls are sharing resources that must be serialized, these operations should be dispatched and executed concurrently. Even if hardware support is not available for parallel execution, it may be possible to execute events and requests concurrently by overlapping CPU-intensive operations with I/O-intensive operations.

Problem: The Monitor Object pattern is unsuited for implementing concurrent dispatching upcalls since it serializes all operations on the dispatching table.

Forces: Although dispatching table modifications typically require exclusive access, dispatching operations do not modify the table. However, the dispatching mechanism must ensure that the table is not modified while a thread is performing a lookup operation on it.

Solution: Use a readers/writer lock to serialize access to the dispatching table. The critical path, *i.e.*, looking up the target object and invoking an operation on it, does not modify the table. Therefore, a `read` lock will suffice for this path. Operations that modify the dispatching table, such as adding or removing objects from it, require exclusive access. Therefore, a `write` lock is required for these operations. Figure 4 illustrates the structure of this solution, where multiple reader threads can simultaneously dispatch operations while writer threads are serialized.

Consequences: Readers/writer locks allow multiple readers to access a shared resource simultaneously while only allowing one writer to access the shared resource at a time. Thus, the solution described above allows multiple concurrent dispatch calls.

Some DOC middleware executes the upcall in a separate thread in the same process or on a remote object. Other middleware executes the upcall in the same thread after releasing the `read` lock. Thus, this pattern can be applied to such systems without any risk of deadlocks. However, this solution is not applicable to systems that execute the upcall while

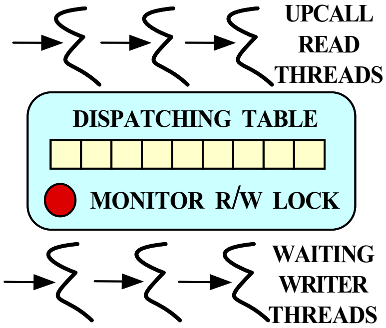

Figure 4: Dispatching with a Readers/Writer Lock

holding the `read` lock. In that case, changing the table from within an upcall would require upgrading the readers/writer lock from a `read` lock to a `write` lock. Unfortunately, standard readers/writer lock implementations, such as Solaris/UI threads [18], do not support upgradable locks. Even when this support exists, lock upgrades will not succeed if multiple threads require upgrades simultaneously.

Note that applications using readers/writer locks become responsible for providing appropriate serialization of their data structures since they cannot rely on the dispatching mechanism itself to serialize upcalls. As with recursive locks, serialization overhead may be higher compared to regular locks [17] when there is little or no contention on the dispatching table.

Implementors of this pattern must analyze their dispatching mechanism carefully to identify operations that require only a `read` lock versus those that require a `write` lock. For example, the CORBA Object Adapter supports activation of objects even from upcalls. Thus, when a dispatch lookup is initiated, the Object Adapter does not know if the upcall will modify the dispatching table. Note that acquiring a `write` lock *a priori* is self-defeating since it may unnecessarily impede concurrent access to the table.

Finally, this solution does not resolve the predictability problem. In particular, unbounded priority inversion may occur when high-priority writer threads are suspended waiting for low-priority reader threads to complete dispatching upcalls.

3.1.4 Reference Counting During Dispatch

Context: As before, the dispatching mechanism is used in a multi-threaded system. However, assume the system has stringent QoS requirements that demand predictable and efficient behavior from the dispatching mechanism.

Problem: To be predictable, the system must eliminate all unbounded priority inversions. In addition, bounded priority inversions must be reduced to maximize the efficiency of the system.

Forces: During an upcall, an application can invoke operations that modify the dispatching table. In addition, the dispatching mechanism must be efficient and scalable, maximizing concurrency whenever possible.

Solution: Reference count the entries of the dispatching table by using a single lock to serialize (1) changes to the referenced count and (2) modifications to the table. As shown in Figure 5, the lock is acquired during the upcall, the appropri-

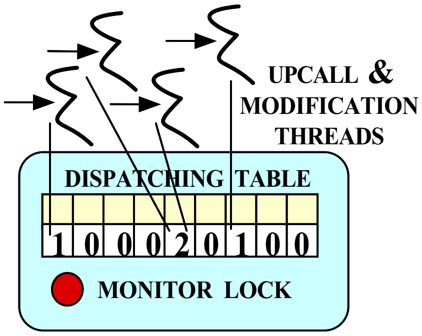

Figure 5: Dispatching with a Reference Counted Table Entries

ate entry is located, its reference count increased, and the lock is released before performing the application upcall. Once the upcall completes, the lock is re-acquired, the reference count on the entry is decremented, and the lock is released.

As long as the reference count on the entry remains greater than zero, the entry is not removed and the corresponding object is not destroyed. Thus, concurrency hazards are avoided because the reference count is always greater than zero while a thread is processing an upcall for that entry. If an object is "logically" removed from the dispatching table, its entry is not "physically" removed immediately since outstanding upcalls may still be pending. Instead, the thread that brings the reference count to zero is responsible for deleting this "partially" removed entry from the table.

In programming languages without garbage collection support, the dispatching table must collaborate with the application to control the objects' life-cycle. In this case, objects are usually reference counted[1] such that the count is incremented when the object is registered with the dispatching table and decremented when the object is removed from the dispatching table.

Consequences: This pattern supports multiple simultaneous upcalls since the lock is not held during the upcall. For the same reason, this model also supports recursive calls . An important benefit of this pattern is that the level of priority inversion does not depend on the duration of the upcall. In fact, priority inversion can be calculated as a function of the

[1]Note that this reference count is different from the per-entry reference count described above.

time needed to search the dispatching table. In our previous research, we have shown that techniques, such as active demultiplexing and perfect hashing [19], incur very low and bounded lookup times. Implementations that use both lookup algorithms described in [19], along with the serialization pattern described here, can achieve predictable dispatching with bounded priority inversions.

A disadvantage of this pattern, however, is that it acquires and releases the lock twice per upcall. In practice, this usually does not exceed the cost of a single recursive monitor lock or a single readers/writer monitor lock [17]. This solution does, however, warrant extra care in the following special circumstances:

- A new request arrives for an object that has been logically, but not physically removed from the dispatching table. Additional state can be used to remember that this object has been removed and should therefore receive no new requests.
- Semantics should be specified for an object that has been partially removed (as described above) and the application requests a new object to be inserted for the same identifier as the partially removed object. Typically, the new insertion would have to block until upcalls on the old object complete and the old object is physically removed from the dispatching table.

3.2 Dispatching to Multiple Targets

The patterns described in this section focus on middleware mechanisms where events or requests are dispatched to multiple target objects. Sending the same event to multiple target objects adds another level of complexity to dispatching mechanism implementations. Typically, an implementation must iterate over the collection of potential targets and invoke upcalls on a subset of the objects in the dispatching table.

In many use-cases, modifications to the collection invalidate any iterators for that collection [20], even for single-threaded configurations. In general, an implementation must ensure that no modifications are performed while a thread is iterating over the dispatching table. Moreover, for real-time systems, simple serialization techniques can result in unbounded priority inversion if higher priority threads wait for lower priority threads to finish iterating.

Interestingly, the most sophisticated pattern for dispatching to a single target object, which was presented in Section 3.1.4, is not suitable for dispatching to multiple targets. In particular, the lock would have to be acquired for the entire iteration and upcall cycle, thereby worsening priority inversion problems. If the lock was released, it could lead to an inconsistent view of the dispatching table. Below, we present a successive series of patterns that address these problems.

3.2.1 Copy then Dispatch

Context: An event or request must be dispatched to multiple objects concurrently.

Problem: The challenge is how to optimize throughput while reducing contention and serialization overhead.

Forces: Modifications to the dispatching table are common during the dispatch loop. The dispatching table does not provide robust iterators [20] or the iterators are not thread-safe. There are no stringent real-time requirements.

Solution: Make a copy of the entire dispatching table before initiating the iteration, as shown in Figure 6. Although some

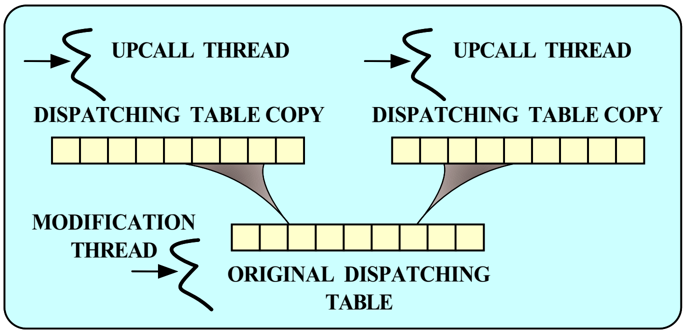

Figure 6: Copy then Dispatch

serialization mechanism must be used during the copy, its cost is relatively low. As an optimization, the dispatching mechanism can acquire the lock, copy only the target objects that are interested in the event, and then release the lock. At this point, the dispatching mechanism iterates over the smaller set of interested target objects and dispatches upcalls.

To apply this pattern, applications must collaborate with the dispatching mechanism to control object life-cycle. For example, an object cannot be destroyed simply because it was removed successfully from the dispatching table. Other threads may still be dispatching events on an older copy of the dispatching table, and thus still have a reference to the object. Therefore, objects in the dispatching table copy must be marked "in use" until all the dispatching loops using it complete.

Consequences: This approach allows multiple events or requests to be dispatched concurrently. In addition, it permits recursive operations from within application upcalls that can modify the dispatching table, either by inserting or removing objects.

However, making copies of the dispatching table does not scale well, when (1) the table is large, (2) memory allocation is expensive, or (3) object life-cycle management is costly. In this case, other patterns, such as the Thread-Specific Storage pattern [21] that reduces locking overhead, can be used to minimize these costs, thereby making this pattern applicable for systems that have small dispatching tables.

3.2.2 Copy On-Demand

Context: As in Section 3.2.1, an event or request must be dispatched to multiple objects concurrently.

Problem: Making copies of the dispatching table is expensive and is not scalable.

Forces: Changes to the dispatching table are infrequent. The dispatching table does not provide robust iterators [20] or the iterators are not thread-safe. In addition, there are no stringent real-time requirements.

Solution: Copy the table on-demand, as shown in Figure 7. When starting an iteration, a counter flag is incremented to indicate that a thread is using the table. If a thread wishes to modify the table it must *atomically* (1) make a copy of the dispatching table, (2) make the modification on the copy, and (3) replace the reference to the old table with a reference to the new one. When the last thread using the original dispatching table finishes its iteration, the table must be deallocated. In programming languages without garbage collection a simple reference count can be used to accomplish this memory allocation strategy.

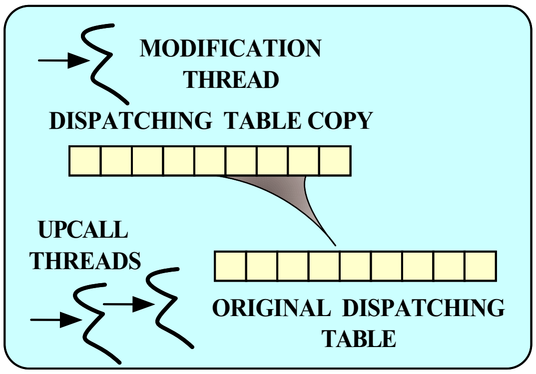

Figure 7: Copy On-Demand

Consequences: Since the solution does not copy the dispatching table when initiating the dispatch loop, this pattern improves the dispatch latency when compared to "Copy then Dispatch" pattern described in Section 3.2.1. Note that locks are not held while executing the upcalls. Thus, an application upcall can invoke recursive operations without risking deadlock.

One downside with this pattern is that it acquires the lock at least twice. The first acquisition occurs when the state is updated to indicate the start of an iteration. The second acquisition indicates the end of the same iteration. Thus, this solution is slightly more expensive than simply holding a lock over the entire dispatch loop when there is little or no contention.

Moreover, when threads contend to initiate a dispatch iteration, some priority inversion may occur. Since the lock is held for a short and bounded period of time, the priority inversion is bounded, however. In contrast, when a thread makes changes to the dispatching table, the amount of time for which it holds the lock depends on the size of the table, which may result in longer priority inversions. Thus, this pattern may be unsuitable for systems with stringent real-time requirements.

3.2.3 Asynchronous Change Commands

Context: An application with stringent QoS requirements where events or requests must be dispatched to multiple objects concurrently.

Problem: Modifications to the dispatching table must be serialized. However, the amount of time locks are held must be bounded to minimize priority inversions.

Forces: Upcalls are executed in the same thread that dispatches the event. The application can add and remove objects from the dispatching table dynamically.

Solution: Postpone changes to the dispatching table while threads are dispatching upcalls. Before iterating over the dispatching table, the dispatching thread atomically increments a counter that indicates the number of threads currently iterating over the dispatching table. When an iteration completes it atomically decrements the counter. If a change is requested while the dispatching table is "busy," the request is converted into a Command Object [8], as shown in Figure 8, and queued to be executed when the dispatching table becomes "idle," *i.e.*, when no more dispatching threads are iterating over the table.

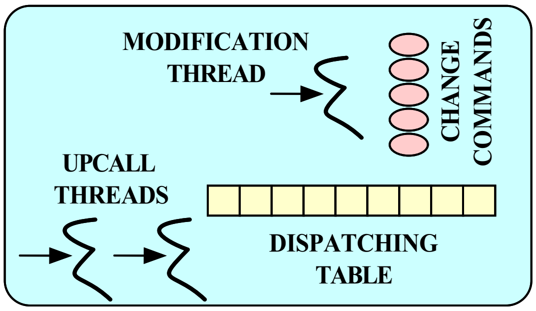

Figure 8: Asynchronous Change Commands

Consequences: Queuing a change to the dispatching table requires a bounded amount of time, thus preventing unbounded priority inversions. For similar reasons, this solution does not deadlock when upcalls request modifications since they are simply queued.

There is, however, a more subtle priority inversion in this implementation. A high-priority thread can request a modification, but the modification will not take place until the potentially lower priority threads have finished dispatching events. In many systems this is an acceptable tradeoff since the critical path, *i.e.*, the dispatching path, is where priority inversions must be avoided.

In addition, it is hard to ascertain when requested modifications actually occur because they execute asynchronously. Likewise, it is hard to report errors when executing change requests because the thread requesting the change does not wait for operations to complete.

4 Concluding Remarks

This paper describes a pattern language for developing and selecting appropriate solutions to common problems encountered when developing efficient, scalable, predictable, and flexible dispatching mechanisms for DOC middleware and applications. Our long term goal is to develop a handbook of patterns for developing real-time DOC middleware. Though we have not completed that goal, patterns have proven quite useful to help middleware researchers and developers reuse successful techniques and practices. Not only do patterns teach useful techniques, they help developers communicate better, and they help developers reason about what they do and why. In addition, patterns are a step toward the handbook for DOC middleware.

The pattern language documented in this paper has been applied to the TAO real-time ORB [3]. TAO has been used for a wide range of real-time applications, including avionics mission computing systems at Boeing [4, 22, 23], the SAIC Run Time Infrastructure (RTI) implementation [5] for the Defense Modeling and Simulation Organization's (DMSO) High Level Architecture (HLA) [24], and high-energy testbeam acquisition systems at SLAC [25] and CERN [26]. The source code and documentation for the TAO ORB and its Event Service are freely available from URL `www.cs.wustl.edu/~schmidt/TAO.html`.

References

[1] C. Alexander, *The Timeless Way of Building*. New York, NY: Oxford University Press, 1979.

[2] Object Management Group, *Realtime CORBA Joint Revised Submission*, OMG Document orbos/99-02-12 ed., March 1999.

[3] D. C. Schmidt, D. L. Levine, and S. Mungee, "The Design and Performance of Real-Time Object Request Brokers," *Computer Communications*, vol. 21, pp. 294–324, Apr. 1998.

[4] T. H. Harrison, D. L. Levine, and D. C. Schmidt, "The Design and Performance of a Real-time CORBA Event Service," in *Proceedings of OOPSLA '97*, (Atlanta, GA), ACM, October 1997.

[5] C. O'Ryan, D. C. Schmidt, and D. Levine, "Applying a Scalable CORBA Events Service to Large-scale Distributed Interactive Simulations," in *Proceedings of the* 5^{th} *Workshop on Object-oriented Real-time Dependable Systems*, (Montery, CA), IEEE, Nov. 1999.

[6] D. C. Schmidt, "Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching," in *Pattern Languages of Program Design* (J. O. Coplien and D. C. Schmidt, eds.), pp. 529–545, Reading, MA: Addison-Wesley, 1995.

[7] I. Pyarali, T. H. Harrison, D. C. Schmidt, and T. D. Jordan, "Proactor – An Architectural Pattern for Demultiplexing and Dispatching Handlers for Asynchronous Events," in *Pattern Languages of Program Design* (B. Foote, N. Harrison, and H. Rohnert, eds.), Reading, MA: Addison-Wesley, 1999.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.

[9] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture - A System of Patterns*. Wiley and Sons, 1996.

[10] I. Pyarali and D. C. Schmidt, "An Overview of the CORBA Portable Object Adapter," *ACM StandardView*, vol. 6, Mar. 1998.

[11] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, 2.3 ed., June 1999.

[12] Object Management Group, *CORBAServices: Common Object Services Specification, Revised Edition*, 95-3-31 ed., Mar. 1995.

[13] D. C. Schmidt, "Strategized Locking, Thread-safe Interface, and Scoped Locking: Patterns and Idioms for Simplifying Multi-threaded C++ Components," *C++ Report*, vol. 11, Sept. 1999.

[14] J. O. Coplien and D. C. Schmidt, eds., *Pattern Languages of Program Design*. Reading, MA: Addison-Wesley, 1995.

[15] D. C. Schmidt, "Monitor Object – an Object Behavior Pattern for Concurrent Programming," *C++ Report*, vol. 12, Mar. 2000.

[16] Paul E. McKinney, "Selecting Locking Designs for Parallel Programs," in *Pattern Languages of Program Design* (J. O. Coplien, J. Vlissides, and N. Kerth, eds.), Reading, MA: Addison-Wesley, 1996.

[17] D. C. Schmidt, "An OO Encapsulation of Lightweight OS Concurrency Mechanisms in the ACE Toolkit," Tech. Rep. WUCS-95-31, Washington University, St. Louis, September 1995.

[18] J. Eykholt, S. Kleiman, S. Barton, R. Faulkner, A. Shivalingiah, M. Smith, D. Stein, J. Voll, M. Weeks, and D. Williams, "Beyond Multiprocessing... Multithreading the SunOS Kernel," in *Proceedings of the Summer USENIX Conference*, (San Antonio, Texas), June 1992.

[19] I. Pyarali, C. O'Ryan, D. C. Schmidt, N. Wang, V. Kachroo, and A. Gokhale, "Applying Optimization Patterns to the Design of Real-time ORBs," in *Proceedings of the* 5^{th} *Conference on Object-Oriented Technologies and Systems*, (San Diego, CA), USENIX, May 1999.

[20] T. Kofler, "Robust iterators for ET++," *Structured Programming*, vol. 14, no. 2, pp. 62–85, 1993.

[21] D. C. Schmidt, T. Harrison, and N. Pryce, "Thread-Specific Storage – An Object Behavioral Pattern for Accessing per-Thread State Efficiently," *C++ Report*, vol. 9, November/December 1997.

[22] Christopher D. Gill et al., "Applying Adaptive Real-time Middleware to Address Grand Challenges of COTS-based Mission-Critical Real-Time Systems," in *Proceedings of the 1st IEEE International Workshop on Real-Time Mission-Critical Systems: Grand Challenge Problems*, Nov. 1999.

[23] B. S. Doerr, T. Venturella, R. Jha, C. D. Gill, and D. C. Schmidt, "Adaptive Scheduling for Real-time, Embedded Information Systems," in *Proceedings of the 18th IEEE/AIAA Digital Avionics Systems Conference (DASC)*, Oct. 1999.

[24] F. Kuhl, R. Weatherly, and J. Dahmann, *Creating Computer Simulation Systems*. Upper Saddle River, New Jersey: Prentice Hall PTR, 1999.

[25] SLAC, "BaBar Collaboration Home Page." http://www.slac.stanford.edu/BFROOT/.

[26] A. Kruse, "CMS Online Event Filtering," in *Computing in High-energy Physics (CHEP 97)*, (Berlin, Germany), Apr. 1997.