11 Communication Middleware

???

This chapter illustrates how we applied a key pattern sequence from the pattern language for distributed computing in part II of this book to develop communication middleware that can be used for—among other things—our warehouse management process control system. This middleware allows clients to invoke operations on distributed objects without concern for object location, programming language, operating system platform, communication protocols and interconnects, and hardware. A novel aspect of our communication middleware is its highly configurable, scalable, and reusable design and implementation, which can be tailored to meet specific application quality of service (QoS) requirements and network/ endsystem characteristics more easily than writing the code manually or using conventional middleware implementations that are hard-coded to a single set of strategies.

11.1 A Middleware Architecture for Distributed Systems

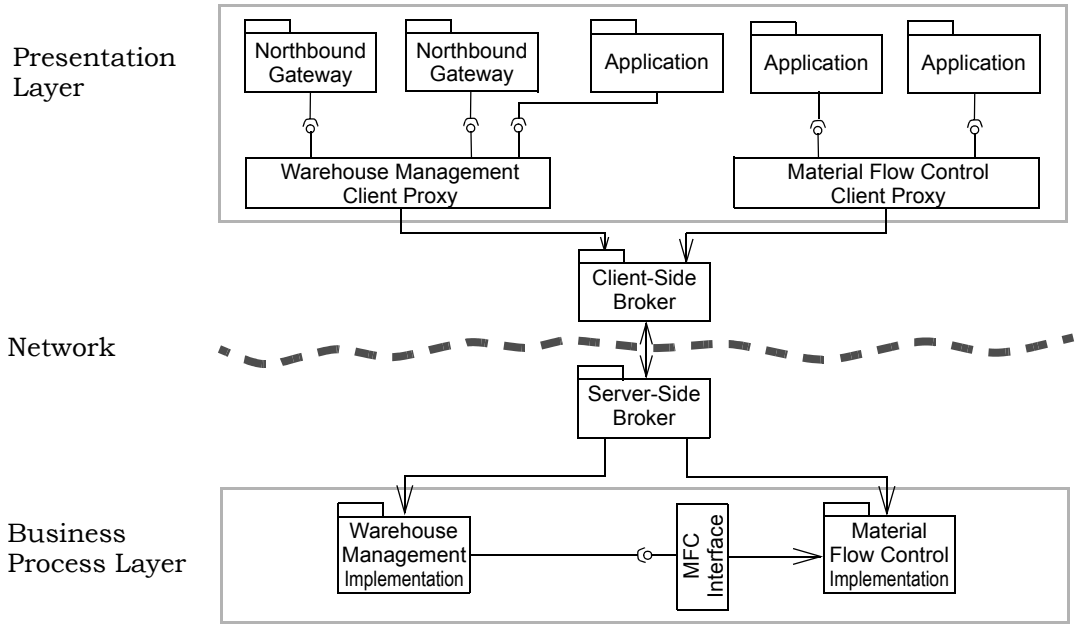
Concrete deployments of our warehouse management process control system typically involve different hardware and software platforms. For instance, client applications and user interfaces are often deployed on Windows PCs, sensor and actuators are usually deployed on embedded devices running VxWorks, and DOMAIN OBJECTS (XYZ) representing business logic and infrastructure functionality are commonly deployed on servers running Solaris or Linux. All devices are connected via different types of network, such as, wireless and wireline LANs and WANs, using a variety of communication protocols, such as TCP/IP, PROFIbus, or VME. Each system installation must also integrate with existing legacy and third-party software, particularly software that resides at the operational and entity level of the automation pyramid, which is often written in a variety of different programming languages, such as C, C++, Java, and C#. The resulting heterogeneity yields development and integration challenges over the system's lifetime, particularly as various software components are removed and/or replaced by components from other vendors.

Communication middleware, such as the *Common Object Request Broker Architecture* (CORBA) [OMG04a] and *Enterprise Java Beans* (EJB) [Sun03] [Sun04], resides between clients and servers in a distributed system. The goal of communication middleware is to simplify application development and integration by providing a uniform view of lower-level—often heterogeneous—network and operating system services. Moreover, this middleware helps off-load complex distributed system infrastructure tasks from application developers to middleware developers [ScSc02], who implement common network programming mechanisms, such as connection management, data transfer, event and request demultiplexing, (de)marshaling, and concurrency control.

To simplify inter-process communication between distributed DOMAIN OBJECTS of the warehouse management process control system, and to shield their implementations from the heterogeneity of its computational environment, the system's base-line architecture uses BROKER-based (238) communication middleware. A BROKER allows distributed DOMAIN OBJECTS to find, access, and communicate with

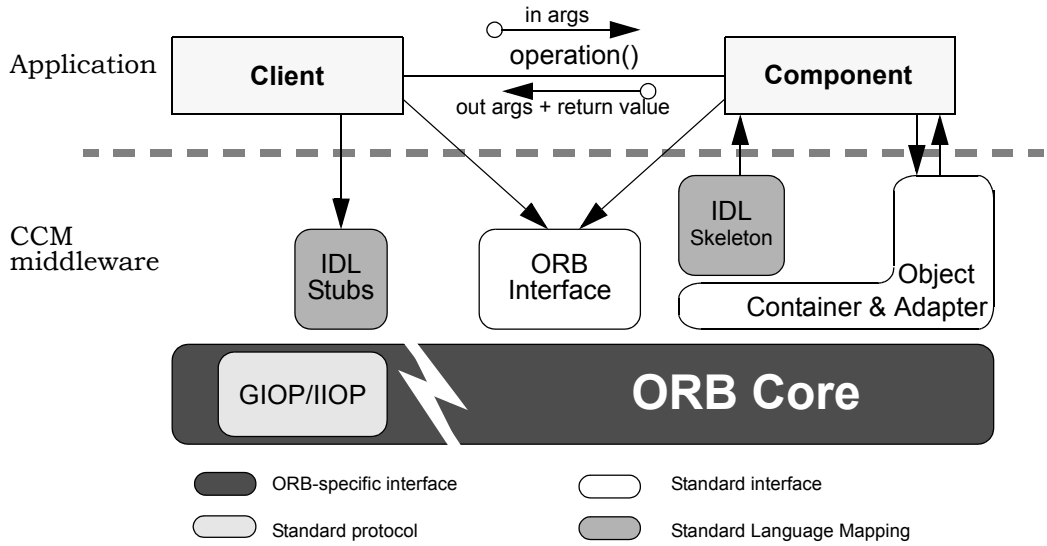
each other as if they were collocated, and decouples them within a distributed system so that they can be developed and integrated using diverse technologies in a heterogeneous environment.

The following diagram illustrates the use of the BROKER-based communication middleware for the presentation layer and the business process layer of the warehouse management process control system. This diagram assumes that DOMAIN OBJECTS within a LAYER are collocated. The remote interaction between other LAYERS of the system or between DOMAIN OBJECTS of the same LAYER is organized correspondingly.



We implement the BROKER communication middleware for our warehouse management process control system using the *CORBA Component Model* (CCM) [OMG02]. The CCM is communication middleware that allows applications to invoke operations on components without concern for their location(s), programming language, operating system platforms, network protocols and interconnects, and hardware. CCM is essentially a language- and platform-independent variant of EJB. At the heart of the CCM

reference model is the BROKER whose elements are shown in the following figure.

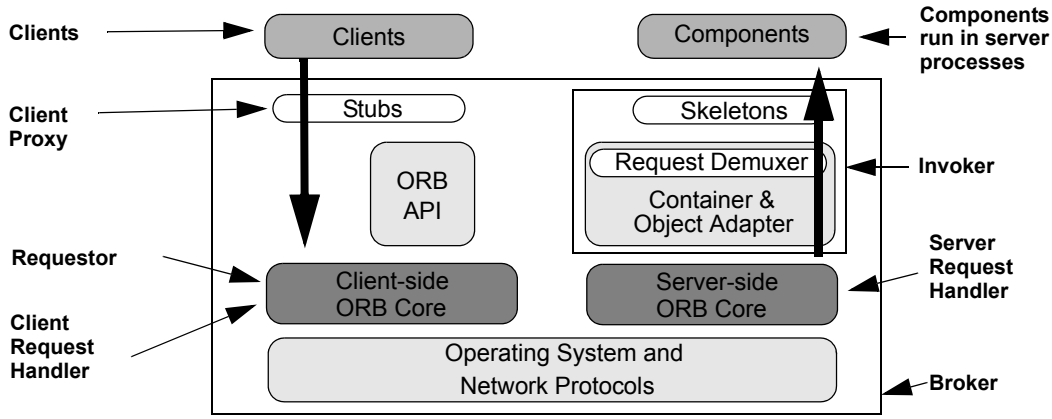


The CCM reference model defines the following key entities:

- *Clients* and *Components* implement the applications running atop the communication middleware.
- An *Object Request Broker (ORB) Core*[1] is responsible for delivering a operation request from a client to a component and returning a response, if any.
- The *ORB Interface* decouples applications from implementation details of the ORB Core.
- *IDL Stubs and Skeletons* serve as a 'glue' between the client and components, respectively, and the ORB.
- A *Container* and *Object Adapte*r associate a component with an ORB by providing a run-time environment for managing component lifecycle properties, demultiplexing incoming requests to the component, and dispatching the appropriate upcall method on that component.

1. The OMG CCM specification uses the term *Object* Request Broker for backwards compatibility with earlier versions of the CORBA specification, even though application developers who use CCM typically program and interact with *Component*s.

The CCM reference model, however, intentionally just specifies the fundamental roles that must be present in an ORB, but does not define a concrete software architecture to use as the basis for a specific CCM implementation. We therefore used the BROKER pattern to define the actual components, their relationships, and their collaborations needed to implement CCM-compliant communication middleware, as shown in the following figure.



The client and server roles in the BROKER pattern represent the application-level clients and components in the OMG CCM reference model. The CLIENT PROXY (281) role is implemented by the ORB's Stubs, which provide access to the services offered by remote components. Stubs also shield clients and servers from the location and the implementation details of their remote communication partners, as well as from the specifics of the ORB implementation.

The Client-side ORB Core maps to the REQUESTOR (XYZ) and CLIENT REQUEST HANDLER (XYZ) roles in the BROKER pattern, and the Server-side ORB Core to the SERVER REQUEST HANDLER (XYZ) role and the Container and Object Adapter map to the INVOKER (XYZ) role [VKZ04]. All these roles are responsible for the location-transparent transmission of requests from clients to servers, as well as the transmission of responses and exceptions from servers back to clients. The Server- and Client-side ORB Core offer APIs to servers and clients for registering components and invoking methods on components, respectively. These APIs represent the ORB Interface of the CCM reference model.

Applying the BROKER pattern to implement a CCM-based ORB required the resolution of a number of design challenges. Chief among these challenges included structuring the ORB's internal design to enable reuse and effective separation of concerns, encapsulating low-level system calls to enhance portability, demultiplexing ORB Core events and managing ORB connections efficiently and flexibly, enhancing ORB scalability by processing requests concurrently and using an efficient synchronized request queue, enabling interchangeable internal ORB mechanisms and consolidating these mechanisms into groups of semantically compatible strategies, and configuring these consolidated ORB strategies dynamically. The remainder of this chapter describes the pattern sequence we applied to resolve these challenges. The resulting communication middleware provides the BROKER platform for our warehouse management process control system, as well as many other distributed systems in other domains.

11.2 Structuring the Internal Design of the Middleware

CCM-based communication middleware has multiple responsibilities, such as providing APIs to clients and components, routing requests from clients to local or remote components, and their responses back to the clients, and initiating the transmission of such requests and responses over the network. The architecture defined by the BROKER (238) pattern separates application logic from communication middleware functionality. A CCM-based ORB itself, however, is far too complex and fungible to be implemented as a single monolithic component. Moreover, its responsibilities cover different levels of abstraction. For example, APIs are at the Application level, component policy management is at the Container level, request routing and concurrency control at the Object Adapter and ORB Core levels, and request (re)transmission at Operating System and Network Protocol levels.

We therefore need to further decompose our CCM-based ORB communication middleware architecture so that it meets the following requirements:

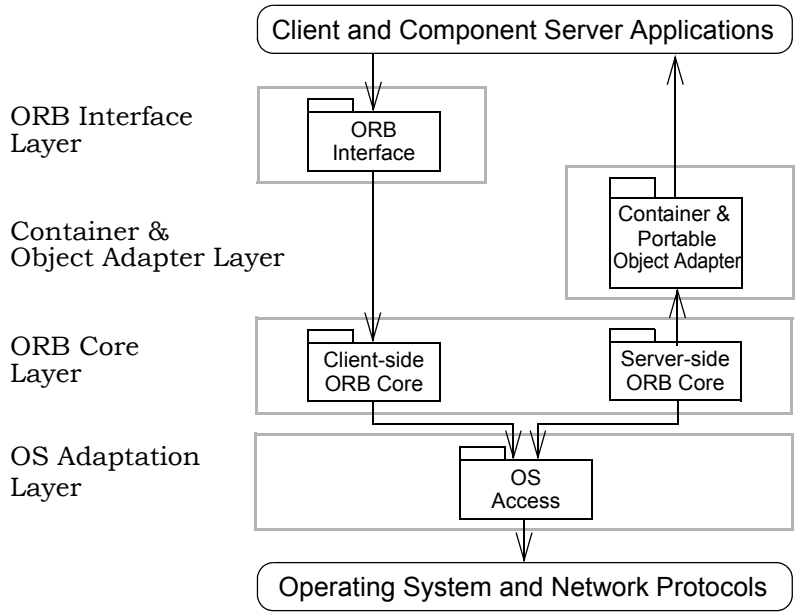
- *Changeability*. Enhancements and changes to one part of the ORB should be confined to a small number of components and not ripple through to affect others.
- *Stability*. External interfaces should be stable, and may even be prescribed by standard OMG specifications, such as the ORB Interface and the various mapping rules for the Interface Definition Language (IDL).
- *Portability*. Porting the ORB to new operating sytem and compiler platforms should affect as little of the ORB as possible. For example, we would like the ORB's transport mechanisms to run on conventional platforms, such as Windows and Linux, as well as small embedded devices, such as sensors and actuators running VxWorks or LynxOS.

How can we decompose the ORB to satisfy the changeability, stability, and portability requirements above and partition its functionality into coherent groups with related responsibilities?

Use LAYERS *(229) to separate different responsibilities in the ORB by decomposing it into groups of subtasks that each handle a particular level of abstraction.*

We divided our CCM-based ORB into four layers. The top layer provides the standard CCM ORB Interface defined by the OMG and represents the 'application view' to the entire ORB. The second layer provides the Container and Object Adapter, which are responsible for managing component policies, as well as demultiplexing and dispatching client requests to component implementations. The third layer includes the ORB Core, which implements the middleware's connection management, data transfer, event demultiplexing, and concurrency control logic. The bottom layer shields the rest of the ORB from the implementation details of the underlying operating system and network protocols.

The diagram below illustrates the layered design for our CCM-based ORB:



Using LAYERS to implement a BROKER enhances its stability because multiple higher-layer client and server applications can reuse the lower-layer services provided by the ORB. Moreover, this design enables transparent changes to implementations at one layer without affecting other layers. For example, the search structures in Object Adapters can be changed from dynamic hashing to active demultiplexing [PRS+00] without affecting any other layers. LAYERS also make it easier to port the ORB to new operating systems requested by customers of our warehouse management process control system without affecting the rest of the ORB or application code.

11.3 Encapsulating Low-level System Mechanisms

One role of communication middleware is to shield client and server applications from operating system and networking details. CCM-based ORB middleware developers—rather than application developers—are therefore responsible for handing these lower-level programming details, such as demultiplexing events, sending and receiving requests across one or more network interfaces, and spawning threads to execute requests concurrently. Developing this layer of middleware can be tricky, however, especially when using low-level system APIs written in languages like C, which yield the following problems:

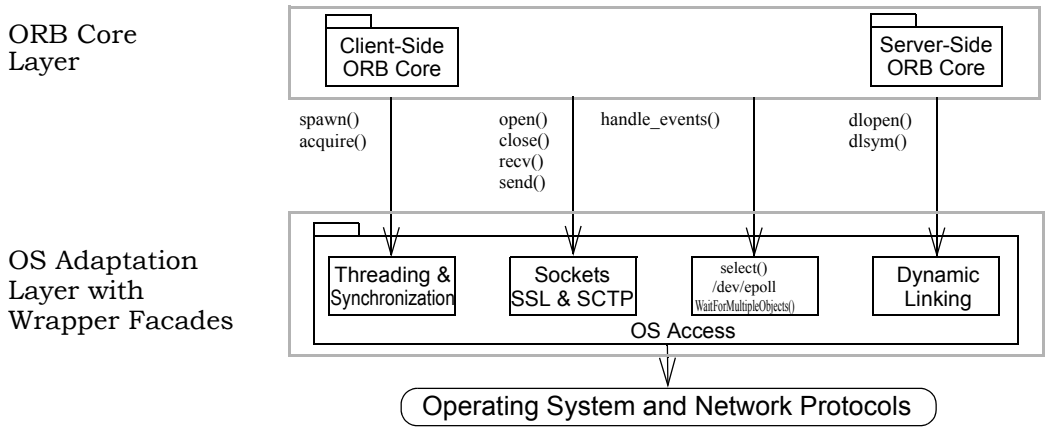
- *ORB developers must have intimate knowledge of many operating system platforms.* Implementing an ORB using system-level C APIs forces middleware developers to deal with the non-portable, tedious, and error-prone operating system idiosyncrasies, such as using weakly-typed socket handles to identify communication endpoints. In addition, these APIs are not portable across operating system platforms. For example, Windows, Linux, and VxWorks have different threading APIs, as well as subtly different semantics for sockets and event demultiplexing.
- *Increased maintenance effort.* One way to build an ORB is to handle portability variations via explicit conditional compilation directives in the ORB source code. Using conditional compilation to address platform-specific variations *at all points of use* increases the complexity of the source code. In particular, it is hard to maintain and extend conditionally compiled code since platform-specific details are scattered throughout the ORB implementation files.
- *Inconsistent programming paradigms.* System mechanisms are accessed through C-style function calls, which cause an 'impedance mismatch' with the object-oriented programming style supported by languages, such as Java, C++, and C#.

How can we avoid accessing low-level system mechanisms directly when implementing ORB middleware?

Structure the OS Adaptation LAYER *(229) of the ORB using* WRAPPER FACADES *(363) to encapsulate system programming APIs and mechanisms within concise and cohesive object-oriented class interfaces.*

A WRAPPER FACADE is a variant of a FACADE (284). FACADE simplifies the interface to a subsystem or framework, whereas WRAPPER FACADE provides type-safe, modular, and portable class interfaces that encapsulate lower-level system and network programming mechanisms, such as sockets, event demultiplexing, synchronization, and threading. In general, WRAPPER FACADE should be applied when existing system-level APIs are non-portable and non-typesafe.

To improve the robustness and portability of our ORB implementation, it accesses all system mechanisms via WRAPPER FACADES provided by the ACE toolkit [SH02], which encapsulate native OS concurrency, communication, memory management, event demultiplexing, and dynamic linking mechanisms with typesafe object-oriented interfaces, as illustrated in the following diagram.



The encapsulation provided by the ACE WRAPPER FACADES provides a consistent object-oriented programming style and alleviates the need for the ORB to access the weakly-typed C system programming APIs directly. Standard compilers and language processing tools can therefore detect type system violations at compile-time rather than at run-time. As a result, much less effort is required to maintain the

ORB, as well as port it to new operating system and compiler platforms requested by customers of our warehouse management process control system.

11.4 Demultiplexing ORB Core Events

An ORB Core is responsible for demultiplexing I/O events from multiple clients and dispatching their associated event handlers. For instance, a server-side ORB Core listens for new client connections and reads General Inter-ORB Protocol (GIOP) REQUEST messages from connected clients, and writes GIOP REPLY messages back to them. To ensure responsiveness to multiple clients, an ORB Core waits for connection, read, and write events to occur on *multiple* socket handles via operating system event demultiplexing mechanisms, such as select(), /dev/epoll, WaitForMultiple Objects(), and threads. Developing this layer of middleware can be tricky, however, due to the following problems:

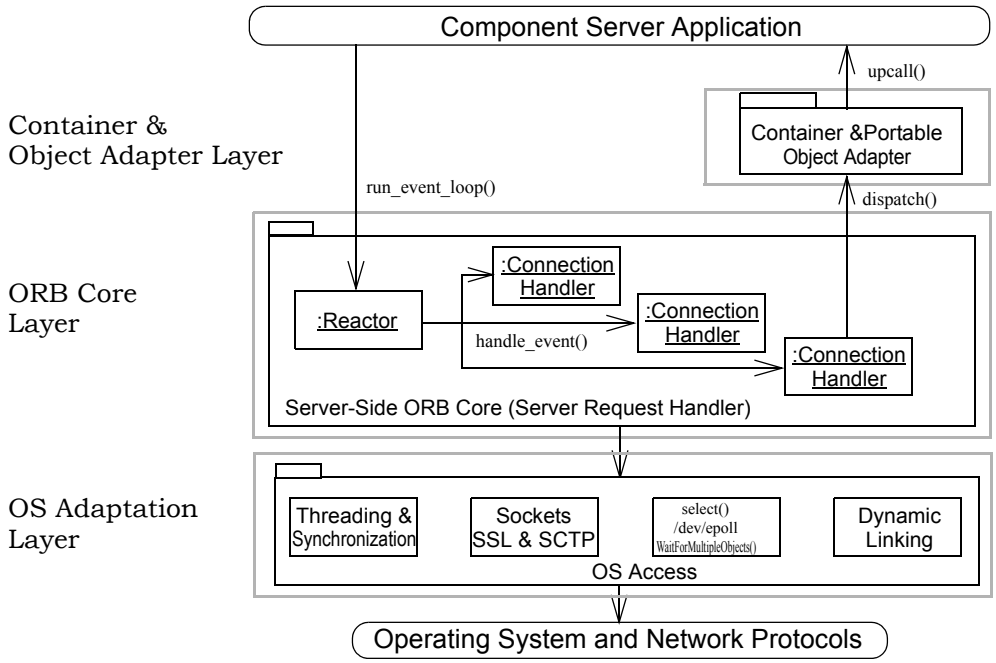
- *Hard-coded event demultiplexers*. One way to develop an ORB is to hard-code it to use a single event demultiplexing mechanism, such as select(). Relying on one event demultiplexing mechanism is undesirable, however, because no single mechanism is the most efficient for *all* platforms and application requirements. For instance, WaitForMultipleObjects() is more efficient than select() on Windows, whereas /dev/epoll is a more efficient demultiplexing mechanism than select() on Linux.
- *Tightly coupled event demultiplexing and event handling code*. Another way to develop an ORB Core is to tightly couple its event demultiplexing code with the code that handles the events, for example, the GIOP protocol processing code. In this case, however, the demultiplexing code cannot be reused as a black box component by other communication middleware applications, such as web servers [HPS97] or video-on-demand applications [MSS00]. In addition, if new ORB strategies for threading or request scheduling algorithms are introduced, substantial portions of the ORB Core must be re-written.

How can an ORB implementation decouple itself from a single event demultiplexing mechanism and decouple its demultiplexing code from its event handling code?

Use a REACTOR *(335) to reduce coupling and increase the extensibility of an ORB Core by supporting demultiplexing and dispatching of multiple event handlers, which are triggered by events that can arrive concurrently from multiple clients.*

A REACTOR simplifies event-driven applications and communication middleware by integrating the demultiplexing of events and the dispatching of their corresponding event handlers. In general, a REACTOR should be introduced when applications or middleware components must handle events from multiple clients concurrently, without becoming tightly coupled to a single low-level mechanism, such as select().

One way to implement an ORB is to use a REACTOR to drive the main event loop within its ORB Core, which plays the SERVER REQUEST HANDLER (XYZ) role of the BROKER (238) architecture, as shown in the following figure of the server-side ORB.



In this design, a component server process initiates an event loop in the ORB Core's Reactor instance, where it remains blocked on whichever event demultiplexing mechanism is configured until I/O events occur on one or more of the available endpoints. When a GIOP REQUEST event occurs, the Reactor demultiplexes the request to the appropriate event handler, which is an instance of the GIOP ConnectionHandler class that is associated with each connected socket. The Reactor then calls the handle_event() method on the ConnectionHandler, which reads the request and passes it to the ORB's Container and Object Adapter layer. This layer then demultiplexes the request to the appropriate upcall method on the component and dispatch the upcall method.

Using the REACTOR pattern enhances the extensibility of the ORB by decoupling the event handling portions of its ORB Core from the underlying operating system event demultiplexing mechanisms. For example, the WaitForMultipleObjects() event demultiplexing system call can be used on Windows, whereas select() or /dev/epoll can be used on UNIX platforms. REACTOR also simplifies the integration of new event handlers. For example, adding a new connection handler that uses the PROFIbus protocol to communicate with non-CCM portions of our warehouse management system does not affect the implementation of the Reactor class.

11.5 Managing ORB Connections

Connection management is another key responsibility of an ORB Core. For instance, an ORB Core that implements GIOP must establish TCP connections and initialize the protocol handlers for each TCP server endpoint. By localizing connection management logic in the ORB core, application components can focus solely on processing application-specific requests and replies, rather than dealing with low-level operating system and network programming tasks.

An ORB Core is not *limited* to running over GIOP and TCP transports, however. For instance, while TCP transfers GIOP requests reliably, its flow control and congestion control algorithms preclude its use for warehouse management sensors and actuators with strigent timeliness properties, where the Streaming Control Transmission Protocol (SCTP) or Real-Time Protocol (RTP) may be more appropriate. Likewise, it may be more efficient to use a shared memory transport mechanism when clients and components are co-located on the same endsystem. Moreover, to protect the integrity and confidentiality of the data, it may be necessary to exchange requests and responses over an encrypted Secure Socket Layer (SSL) connection. An ORB Core should therefore be flexible enough to support multiple transport mechanisms.

The CCM reference architecture explicitly decouples the connection management tasks performed by an ORB Core from the request processing performed by application components. A common way to implement an ORB's *internal* connection management, however, is to use low-level network APIs, such as sockets. Likewise, the ORB's connection establishment protocol is often tightly coupled with it communication protocol. Unfortunately, this design hard-codes the ORB's connection management implementation with the socket network programming API and the TCP/IP connection establishment protocol with GIOP, thereby yielding the following two problems:

- *Inflexibility*. If an ORB's connection management data structures and algorithms are too closely intertwined, substantial effort is required to modify the ORB Core. For instance, tightly coupling the ORB Core to use the socket API makes it hard to change the underlying transport mechanism to use shared memory or SSL rather than sockets. It can therefore be time consuming to port a tightly coupled ORB Core to new networking protocols and programming APIs, such as SSL, SCTP, RTP, or Windows Named Pipes.
- *Inefficiency*. Many internal ORB strategies can be optimized by allowing ORB and application developers to choose appropriate implementations late in the design cycle, such as after conducting extensive run-time performance profiling. For instance, a multi-threaded real-time client may need to store transport endpoints using THREAD-SPECIFIC STORAGE (326). Similarly, the concurrency

strategy for a CCM component server might require that each connection run in its own thread to eliminate per-request locking overhead. If the connection management mechanism is hard-coded and tightly bound with other internal ORB strategies, however, it is hard to accommodate efficient new mechanisms without significant effort and rework.

How can an ORB Core's connection management components support multiple transports and allow connection-related behaviors to be (re-) configured flexibly at any point in the development cycle?

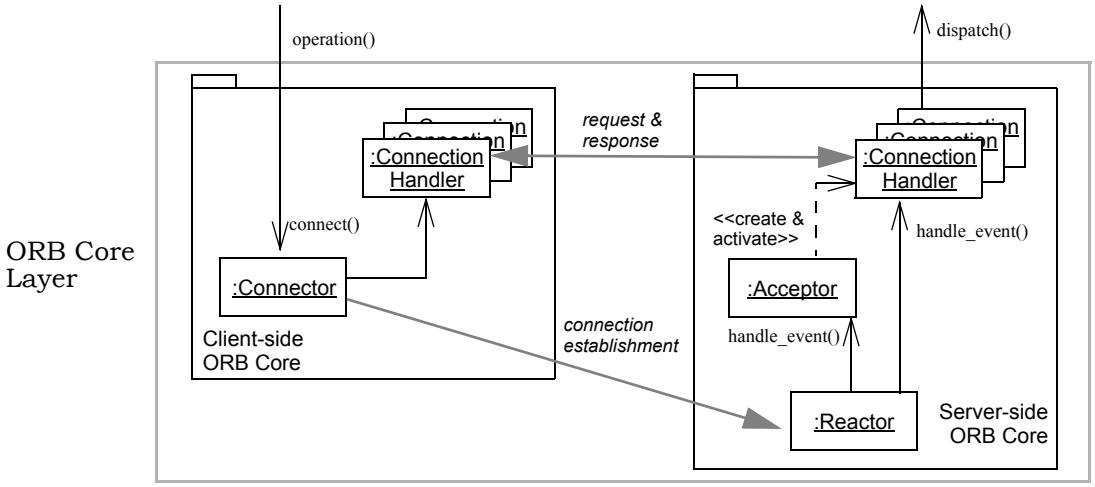
Use an ACCEPTOR-CONNECTOR *(344) configuration to increase the flexibility of ORB Core connection management and initialization by decoupling connection establishment and service initialization from the tasks performed once these activities have completed.*

The acceptor component in the ACCEPTOR-CONNECTOR pattern is responsible for *passive* connection establishment and service initialization, which is performed by the server-side of the ORB Core. Conversely, the connector component in the pattern is responsible for *active* connection establishment and service initialization, which is performed by the client-side of the ORB Core.

Our ORB uses the ACCEPTOR-CONNECTOR pattern in conjunction with the REACTOR pattern (335) to create a pluggable protocols framework [OKS+00]. This framework performs connection establishment and connection handler initialization for the various networking protocols supported in the ORB, as follows:

- *Client-side ORB Core*. In response to an operation invocation or an explicit binding to a remote component, the client-side ORB Core uses a Connector to initiate a connection for the desired type of protocol to the designated server ORB and then initialize the appropriate type of ConnectionHandler to service this connection when it completes.
- *Server-side ORB Core*. When a connection arrives from a client, the server-side ORB Core uses an Acceptor to create the appropriate type of ConnectionHandler to service each new client connection.

Acceptors and Connectors are both event handlers that can be dispatched automatically by the ORB's REACTOR when events become ready for processing, as shown by the following figure.



This figure shows that when a client invokes a remote operation(), it makes a connect() call via its Connector to obtain a connection and initialize a ConnectionHandler that corresponds to the desired type of networking protocol. In the server-side ORB Core, the Reactor notifies an Accceptor via its handle_event() method to accept the newly connected client and create the corresponding ConnectionHandler. After this ConnectionHandler is activated within the ORB Core, it performs the requisite protocol processing on a connection and ultimately dispatches the request to the appropriate component via the ORB's Container and Object Adapter.

The combined use of ACCEPTOR-CONNECTOR and REACTOR in our CCM-based ORB increased its flexibility by decoupling event demultiplexing from connection management and protocol processing. This design also simplified the integration of networking protocols and network programming APIs that are most suitable for particular configurations of our warehouse management process control system.

11.6 Enhancing ORB Scalability

Achieving scalable end-to-end performance is important to handle heavy traffic loads as the number of clients increases our warehouse management process control system. By default, GIOP runs over TCP, which uses flow control to ensure that senders do not produce data more rapidly than slow receivers or congested networks can buffer and process [Ste93]. If a CCM sender transmits a large amount of data over TCP faster than a receiver can process it, therefore, the connection will flow control and block the sender until the receiver can catch up.

Our initial REACTOR design outlined in Section 11.4, *Demultiplexing ORB Core Events*, processed all requests within a single thread of control. Although this design is straightforward to implement, it has the following problems:

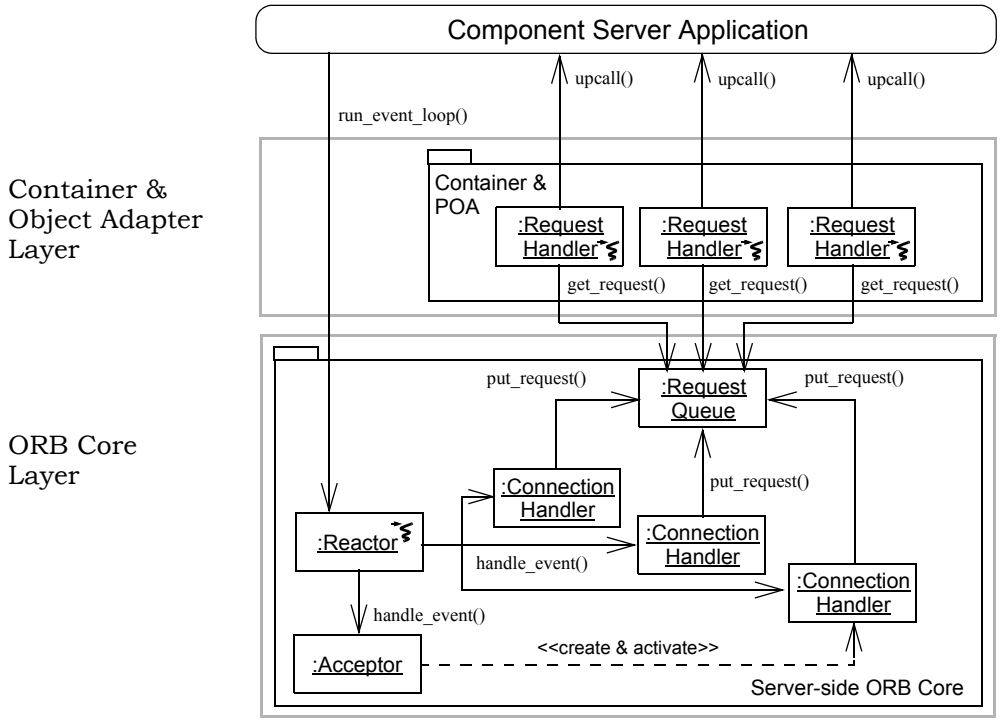
- *Non-scalable*. Processing long-duration client requests reactively within a single-threaded reactive ORB server process scales poorly because only one client request can be handled at a time.
- *Starvation*. The entire ORB server process can block indefinitely while waiting for flow control on a connection to abate when sending a reply to a client, which will starve other clients from having their requests processed.

Conversely, however, multi-threading all ORB processing is also problematic for short-duration processing because threads may incur unnecessary concurrency control overhead, in terms of synchronization, context switching, and data movement [PSC+01].

How can an ORB manage concurrent processing efficiently so that long-running requests can execute simultaneously on one or more CPUs without impeding the progress of other requests, while short-duration processing is handled efficiently without incurring unnecessary concurrency control overhead?

Apply the HALF-SYNC/HALF-ASYNC pattern (299) to separate the short- and long-duration processing in the ORB, thereby enhancing scalability without incurring excessive concurrency control overhead.

The HALF-SYNC/HALF-ASYNC concurrency model for our CCM-based ORB uses a pool of RequestHandlers to process long-duration clients requests and replies concurrently in separate threads of control. Conversely, short-duration Acceptor connection establishment and REQUEST event handling is processed *reactively* in ConnectionHandlers by borrowing the Reactor's thread of control. The following figure illustrates the HALF-SYNC/HALF-ASYNC design of our ORB.



This figure shows how Acceptor connection establishment is driven entirely by the Reactor when it dispatches the Acceptor's handle_event() method. REQUEST event handling is driven partially by the Reactor, which dispatches the ConnectionHandler's handle_event() method to read the request message into a buffer. This buffer is then placed on a synchronized RequestQueue, which is used to pass requests to a pool of RequestHandlers that process the requests concurrently in separate threads of control.

The use of HALF-SYNC/HALF-ASYNC in our ORB improves its scalability compared with using a purely REACTOR-based design by allowing multiple client requests/replies to run concurrently in separate threads. Likewise, because each thread can block independently, the entire server ORB process need not wait for flow control on a particularly connection to abate when sending a reply to a client. Certain subsystems in our warehouse management process control system are better suited by a REACTOR-based design, however, so our ORB supports both approaches.

11.7 Implementing a Synchronized Request Queue

At the center of HALF-SYNC/HALF-ASYNC (299) is a queueing layer. In our CCM-based ORB, the ConnectionHandlers in the asynchronous (reactive) layer are 'producers' that inserts client requests into the RequestQueue. The pool of RequestHandlers in the synchronous (multi-threaded) layer are 'consumers' that remove and process client requests from the queue.

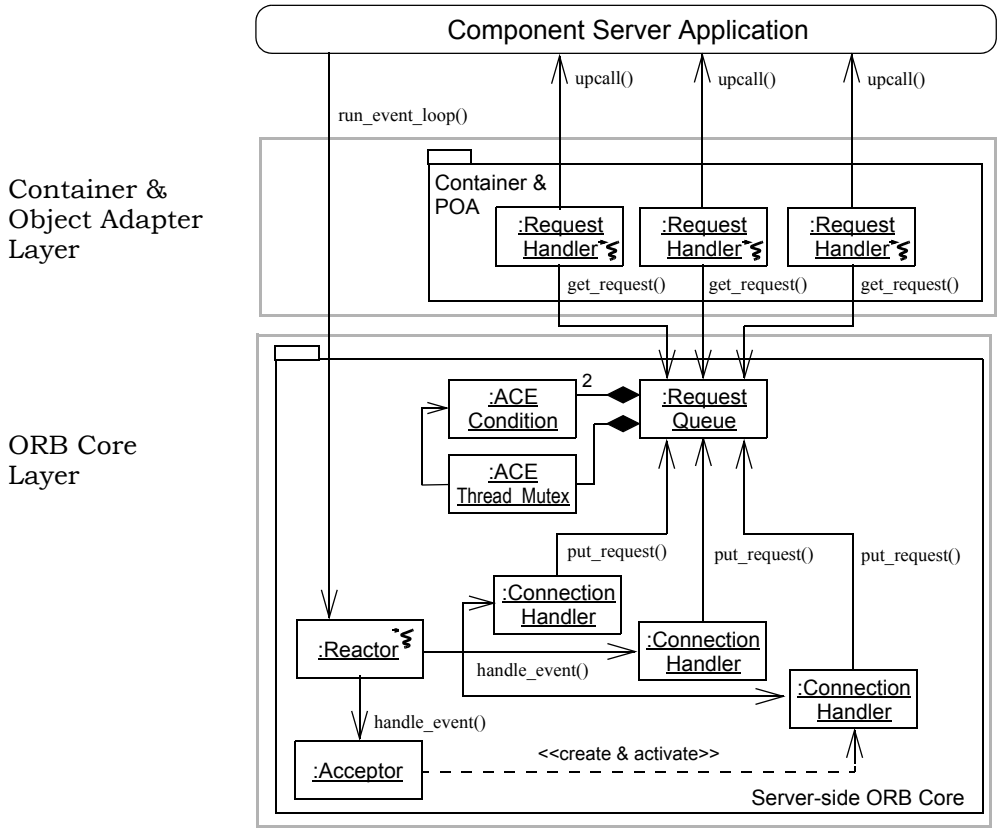
A naive implementation of a RequestQueue can incur several problems. For example, multiple concurrent producer and consumer ORB threads at the different HALF-SYNC/HALF-ASYNC layers can corrupt the RequestQueue's internal state if it is not synchronized properly. Similarly, these threads will 'busy wait' when the queue is empty or full, which wastes CPU cycles unnecessarily.

How can the RequestQueue *avoid race conditions or busy waiting when threads in different layers put and get client requests simultaneously?*

Apply MONITOR OBJECT *(309) to implement a synchronized* RequestQueue *that ensures only one method runs at a time and allows its* put_request() *and* get_request() *methods to schedule their execution sequences co-operatively to prevent producer and consumer threads from busy waiting when the* RequestQueue *is full or empty, respectively.*

The synchronized RequestQueue uses an ACE_Thread_Mutex as the monitor lock to serialize access to the monitor object and a pair of ACE_Condition objects to implement the queue's not-empty and not-full monitor conditions. ACE_Condition is a WRAPPER FACADE for POSIX

condition variables [Lew95] that allows threads to coordinate and schedule their processing efficiently. This synchronized RequestQueue can be integrated into the HALF-SYNC/HALF-ASYNC implementation in the ORB as shown in the following figure.



When a consumer thread running in the pool of RequestHandlers attempts to get a client request from an empty RequestQueue, the queue's get_request() method atomically releases the monitor lock and the thread suspends itself on the not-empty monitor condition. It remains suspended until the RequestQueue is no longer empty, which happens when a ConnectionHandler running in the producer thread puts a client request into the queue.

MONITOR OBJECT simplifies our HALF-SYNC/HALF-ASYNC concurrency design by providing a concise programming model for sharing the

RequestQueue among cooperating threads where object synchronization corresponds to method invocation. The synchronized get_request() and get_request() methods use the RequestQueue's monitor conditions to determine the circumstances under which they should suspend or resume their execution.

11.8 Interchangeable Internal ORB Mechanisms

Communication middleware is often required to support a wide range of application requirements in a wide range of operational environments. To satisfy these different requirements and environments, an ORB may therefore need to support multiple implementations of its internal mechanisms. Examples of such mechanisms include alternative concurrency models, event and request demultiplexers, connection managers and data transports, and (de)marshaling schemes.

One way to support multiple implementations of an ORB's internal mechanisms is to statically configure the ORB at compile-time using preprocessor macros and conditional compilation. For example, since /dev/epoll and WaitForMultipleObjects() are only available on certain operating systems, the ORB source code can be interpersed with #ifdef … #elif … #else … #endif conditional compilation blocks. The value of macros examined by the preprocessor can then be used to choose the appropriate event demultiplexer mechanisms during compilation. Although many ORBs use this approach, it has the following problems:

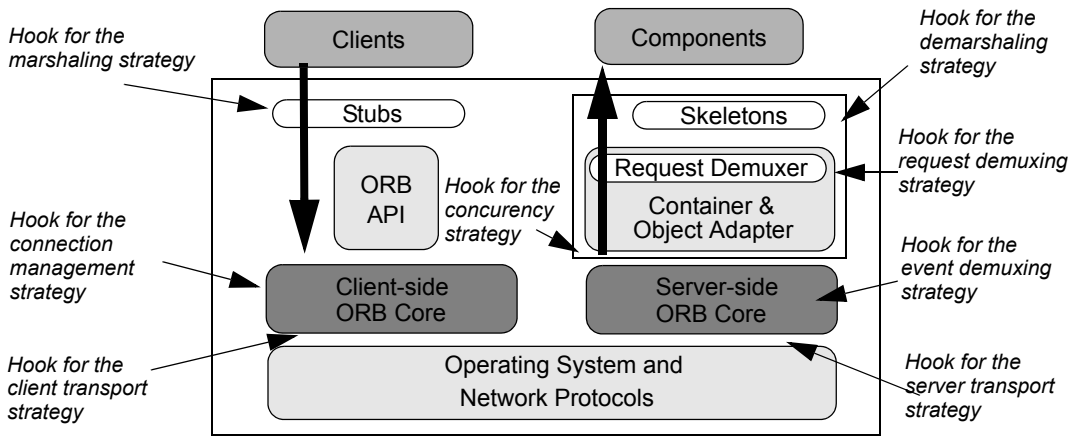
- *Inflexible*. Preprocessor macros can only configure mechanisms known statically at compile-time, which makes it hard to configure an ORB to support mechanisms selected based on knowledge available dynamically during startup or run-time. For example, an ORB might want to configure itself to use different concurrency models or transport mechanisms depending on dynamically discoverable factors, such as the number of CPUs, current workload, or availability of certain networking protocols.

- *Error-prone*. Using preprocessor macros and condition compilation makes it hard to understand and validate the ORB. In particular, changes to the behavior and state of the ORB tend to permate through its source code haphazardly since it is hard to compile and test all paths through the code [MPY+04].

How can an ORB permit replacement of its internal mechanisms in a more flexible manner and encapsulate the state and behavior of each mechanism so that changes to one do not permeate throughout an ORB haphazardly?

Use STRATEGIES *(362) to support multiple transparently 'pluggable' ORB mechanisms by factoring out commonality among alternatives and explicitly associating the name of a strategy with its behavior and state.*

Our CCM-based ORB uses a variety of STRATEGIES to factor out internal mechanisms that are often hard-coded in conventional ORBs. The figure below illustrates where our ORB provides STRATEGY hooks that simplify the configuration of different mechanisms for (de)marshaling, request and event demuxing, connection management and client/server data transport, and concurrency.



Using STRATEGY in our CCM-based ORB removes lexical dependencies on the ORB's internal mechanism implementations since the configured mechanisms are only accessed via common base class interfaces. Moreover, STRATEGY simplifies the customization of ORB

behavior using mechanisms that can be configured *dynamically*, either during startup time or later during run-time, rather than only *statically* at compile-time. Our warehouse management process control system will leverage this capability in various ways, as discussed below.

11.9 Consolidating ORB Strategies

Our CCM-based ORB supports a wide range of strategies in its various layers:

- The Stubs and Skeletons support various *(de)marshaling strategies*, such as the Common Data Representation (CDR), the eXternal Data Representation (XDR), and other proprietary strategies that are only suited for ORBs that communicate across homogeneous hardware, OS, and compilers.
- The Container and Object Adapter layer supports multiple *request demultiplexing strategies*, such as dynamic hashing, perfect hashing, or active demultiplexing [GS97a], and *lifecycle strategies*, such as session containers or entity containers.
- The ORB Core layer supports a variety of *event demultiplexing strategies*, such as REACTORS (335) implemented with select(), /dev/epoll, WaitForMultipleObjects(), or VME-specific demuxers, *connection management strategies*, such as process-wide cached connections versus thread-specific cached connections, ConnectionHandler *concurrency strategies*, such as single-threaded reactive or multi-threaded HALF-SYNC/HALF-ASYNC (299), and different *transport strategies*, such as TCP/IP, SSL, SCTP, VME, and shared memory.

The table below illustrates the strategies used to create two configurations of our ORB for different subsystems in our warehouse management process control system: one for sensor and actuators deployed on embedded devices running VxWorks and the other for

warehouse business logic and infrastructure functionality deployed on servers running Solaris or Linux.

Application	Concurrency Strategy	Marshaling & Demarshaling Strategy	Request Demuxing Strategy	Protocol	Event Demuxing Strategy
Sensors and Actuators	Reactive	Proprietary	Perfect hashing	VME backplane	VME-specific demuxer
Warehouse business logic	HALF-SYNC/ HALF-ASYNC	CDR	Active Demuxing	TCP/IP	select()-based demuxer

Using STRATEGIES so extensively in our ORB, however, can cause it to become overly complex for the following reasons:

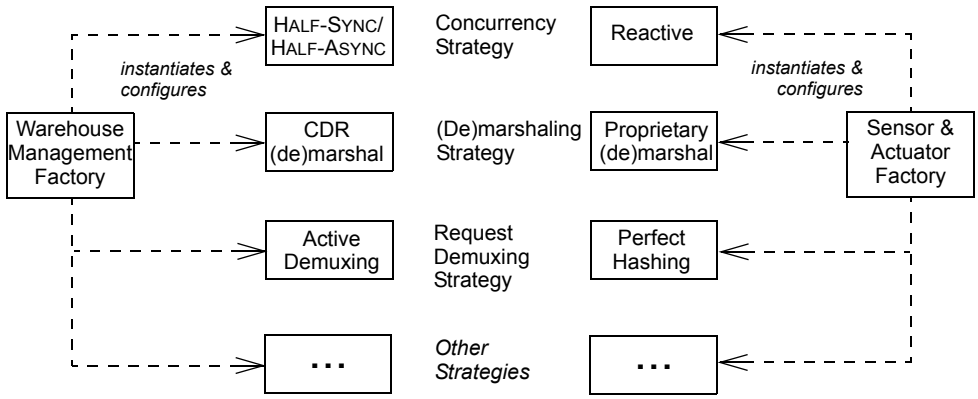
- *Complicated maintenance and configuration.* ORB source code can become littered with hard-coded references to STRATEGY classes, which make it hard to maintain and configure. For example, within a particular subsystem in our warehouse management process control system, such as sensors and actuators or business logic, many independent strategies must act in harmony. Identifying these strategies individually by name, however, requires tedious replacement of selected strategies in one domain with a potentially different set of strategies in another domain.
- *Semantic incompatibilities.* It is not always possible for certain strategies to interact in semantically compatible ways. For instance, the VME-specific event demultiplexing strategy will not work properly with the TCP/IP protocol. Moreover, some strategies are only useful when certain preconditions are met. For instance, the perfect hashing demultiplexing strategy is only applicable to systems that statically register all their components off-line [GS97b].

How can a highly-configurable ORB reduce the complexities required to manage its myriad of strategies, as well as ensure semantic compatibility when combining groups of strategies?

Introduce ABSTRACT FACTORIES *(457) to consolidate multiple ORB strategies into a manageable number of semantically compatible configurations.*

All of our ORB strategies are consolidated into ABSTRACT FACTORIES that encapsulate all the client- and server-specific strategies described above. By using ABSTRACT FACTORY, application developers and end-users can configure the internal mechanisms that comprise different types of ORBs with semantic consistency by providing a single access point that integrates all strategies used to configure an ORB. Concrete subclasses then aggregate semantically compatible application-specific or domain-specific strategies, which can be replaced *en masse* in meaningful ways.

The following figure illustrates two concrete instances of ABSTRACT FACTORY used to configure ORBs for applications running in the business logic or sensor and actuator subsystems of our warehouse management process control system.



Our use of ABSTRACT FACTORY simplifies ORB maintenance and configuration by consolidating groups of ORB strategies with multiple alternative implementations that must vary together to ensure semantic compatibility for different warehouse management process control system subsystems.

11.10Dynamic Configuration of ORBs

The cost of many computing resources, such as memory and CPUs, continue to decrease. ORBs must often still avoid excessive consumption of these finite system resources, however, particularly for real-time and embedded systems that require small memory footprints and predictable CPU processing overhead [GS98]. Likewise, many applications can benefit from an ability to extend ORBs *dynamically* by allowing the configuration of their strategies at run-time.

Although STRATEGY (362) and ABSTRACT FACTORY (457) make it easier to customize ORBs for specific application requirements and system characteristics in semantically compatible configurations, these patterns can still cause the following problems:

- *Excessive resource utilization.* Widespread use of STRATEGY can substantially increase the number of internal mechanisms configured into an ORB, which in turn can increase the system resources required to run the ORB and its applications.
- *Unavoidable system downtime.* If strategies are configured statically at compile-time and/or static link-time using ABSTRACT FACTORY, it is hard to enhance existing strategies or add new strategies without *changing* the existing source code for the consumer of the strategy or the abstract factory, *recompiling* and *relinking* an ORB, and *restarting* running ORBs and their application components to update them with the new capabilities.

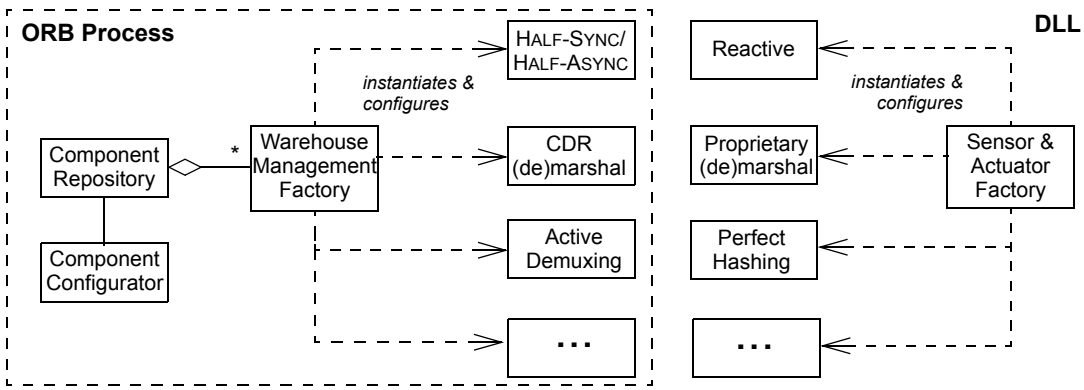
In general, static configuration is only feasible for a small, fixed number of strategies. Using this technique to configure more sophisticated, extensible ORBs complicates maintenance, increases system resource utilization, and requires system downtime to add or change existing components.

How can an ORB implementation reduce the 'overly-large, overly-static' side-effect of pervasive usage of STRATEGY *and* ABSTRACT FACTORY*?*

Introduce a COMPONENT CONFIGURATOR *(418) to dynamically link/unlink custom* STRATEGY *and* ABSTRACT FACTORY *objects into an ORB at startup- or run-time.*

We use a COMPONENT CONFIGURATOR in our CCM-based ORB to configure ABSTRACT FACTORIES at run-time that contain the desired group of semantically compatible STRATEGIES. The ORB's initialization code uses the explicit dynamic linking mechanisms provided by the operating system and encapsulated by the WRAPPER FACADES (363) in the ORB's OS Adaptation Layer to link in the appropriate factory for a particular use case. Commonly used explicit dynamic linking mechanisms include the dlopen()/dlsym()/dlclose() system functions in UNIX and the LoadLibrary()/GetProcAddress() system functions in Windows. By using a COMPONENT CONFIGURATOR in conjunction with these system functions, the *behavior* of the ORB can be decoupled from *when* implementations of its internal mechanisms are configured as semantically compatible STRATEGIES into an ORB.

ORB strategies can be linked into an ORB from dynamic link libraries (DLLs) at compile-time, startup-time, or even later during run-time. Moreover, a COMPONENT CONFIGURATOR can reduce the memory footprint of an ORB by allowing application developers to dynamically link only those strategies that they need for to configure the ORB for their particular use cases. The figure below shows two factories tuned for either the business logic or the sensor and actuator subsystems of our warehouse management process control system.



In this particular configuration, the Sensors and Actuators Factory is currently installed into the ORB's process. Applications using this ORB configuration will therefore be processed with the designated set of ORB concurrency, (de)marshaling, and request demultiplexing strategies, among others. In contrast, the Warehouse Business Logic

Factory resides in a DLL outside of the current ORB process. By using the COMPONENT CONFIGURATOR, this factory could be installed dynamically when the ORB process starts to run.

Within the ORB process, the Sensors and Actuators Factory is maintained by a ComponentRepository that manages all currently loaded configurable components in the ORB. The ComponentConfigurator uses the ComponentRepository to coordinate the (re)configuration of components, for instance, by linking an optimized version of the Sensors and Actuators Factory and unlinking the current version.

COMPONENT CONFIGURATOR allows application developers to configure the behavior of our ORB dynamically to tailor the ORB to meet their specific operational environments and application requirements. In addition to enhancing flexibility, it also ensures that the ORB does not incur the time and space overhead for strategies it does not use. Moreover, this pattern allows application developers to configure the ORB without requiring access to—or modifications of—the ORB source code, and often without having to shutdown the entire ORB to upgrade portions of its behaviors.

11.11 Communication Middleware Summary

The CCM-based ORB design described in this chapter employs a pattern sequence to specify fundamental ORB mechanisms, such as concurrency, transports, request and event demultiplexing, and (de)marshaling, in a well-defined and time-proven manner. Key design goals were to keep the ORB configurable, extensible, adaptable, and portable. The patterns in the sequence used to create this design were selected, integrated, and implemented to achieve these goals based on our extensive experience applying these patterns in other *standard middleware*, such as web servers [POSA2] [HMS97], *object-oriented network programming frameworks*, such as ACE [SH03], and *networked applications*, such as application-level gateways [Sch00b] and electronic medical imaging systems [PHS96].

The first two patterns in the sequence, BROKER (238) and LAYERS (229) define the core structure for our CCM-based ORB. BROKER separates

application functionality from communication middleware functionality, whereas LAYERS separates different communication middleware services according to their level of abstraction.

The third pattern in the sequence, WRAPPER FACADE (363) helps to structure the lowest layer in the ORBs design, the OS Abstraction Layer, into modular and independently usable building blocks. Each WRAPPER FACADE provides a meaningful abstraction for a specific responsibility and/or group of functionality supported by an operating system, and encapsulates the corresponding API functions into a type-safe, modular, and portable class. Higher layers of the ORB can thus be implemented without having explicit dependencies on a specific operating system.

The next set of patterns in the sequence focus on the Container and Object Adapter and ORB Core layers in server-side ORB. In terms of the BROKER architecture, the server-side ORB plays the role of the SERVER REQUEST HANDLER (XYZ), which is responsible for receiving messages and requests from the network and dispatching these messages and requests to their intended component for further processing. A REACTOR (335) provides a demultiplexing and dispatching infrastructure for the ORB Core layer that can be extended to handle different event handling strategies and is independent from low-level demultiplexing mechanisms, such as select() and WaitForMultipleObjects(). An ACCEPTOR-CONNECTOR structure (335) leverages the REACTOR by introducing specialized event handlers for initiating and accepting network connection events, thus separating connection establishment from communication in an ORB Core. HALF-SYNC/HALF-ASYNC (299) and MONITOR OBJECT (309) augment the REACTOR so that client requests can be processed concurrently, thereby improving server-side ORB scalability.

The final three patterns in the sequence address configurability. STRATEGY (457) is used wherever there is variability possible for the ORB's mechanisms, such as its connection management, concurrency, and event/request demultiplexing mechanisms. To configure the ORB with a specific set of semantically compatible strategies, the client- and server-side ORB implementations use an ABSTRACT FACTORY (457). These two patterns work together to make it easier to create variants of the ORB that are customized to meet the needs of particular users and application scenarios. A COMPONENT

CONFIGURATOR (418) is used to orchestrate the update the strategies and abstract factories in the ORB without modifying existing code, recompiling or statically relinking existing code, or terminating and restarting an existing ORB and its application components.

The following table summarizes the mapping between specific ORB design challenges and the pattern sequence we used to resolve these challenges.

Pattern	Challenges
BROKER	Defining the ORB's base-line architecture
LAYERS	Structuring ORB internal design to enable reuse and clean separation of concerns
WRAPPER FACADE	Encapsulating low-level system calls to enhance portability
REACTOR	Demultiplexing ORB Core events effectively
ACCEPTOR-CONNECTOR	Managing ORB connections effectively
HALF-SYNC/HALF-ASYNC	Enhancing ORB scalability by processing requests concurrently
MONITOR OBJECT	Efficiently synchronize the HALF-SYNC/ HALF-ASYNC request queue
STRATEGY	Interchanging internal ORB mechanisms transparently
ABSTRACT FACTORY	Consolidating ORB mechanisms into groups of semantically compatible strategies
COMPONENT CONFIGURATOR	Configuring consolidated ORB strategies dynamically

Analyzing this pattern sequence reveals that it helps to design an ORB that is not only suitable to meet the requirements of our warehouse management process control system, but that is also configurable to meet requirements of distributed systems in many other domains. In particular, we have used our pattern sequence to create a product-line architecture for a specific set of technological concerns—namely, communication middleware—within a larger product-line architecture for an application domain—namely, warehouse management process control. The architecture and

implementation of the ORB is thus an extensible and reusable asset that not only meets our immediate needs, but can also be applied productively well beyond the domain of warehouse management.

Consequently, it is no surprise that the pattern sequence described in this chapter forms the basis of the *Component-Integrated ACE ORB (CIAO)* [WSG+03]. CIAO extends *The ACE ORB (TAO)* [SNG+02] to create a QoS-enabled CCM middleware platform by combining:

- *Lightweight CCM* [OMG04b] features, such as standard mechanisms for specifying, implementing, packaging, assembling, and deploying components.
- *Real-time CORBA* [OMG03] [OMG05] features, such as thread pools, portable priorities, synchronizers, priority preservation policies, and explicit binding mechanisms.

CIAO and TAO are open-source (www.dre.vanderbilt.edu) and have been used in many commercial distributed systems, ranging from avionics and vehtronics; factory automation and process control; telecommunication call processing, switching, and network management; and medical engineering and imaging. Many of these systems need real-time QoS support to meet their stringent computation time, execution period, and bandwidth/delay requirements. Due to its flexible, patterns-based design, however, CIAO and TAO are also well-suited for conventional distributed systems that just require 'best-effort' QoS support. Using patterns that focus on both QoS and configurability, including the pattern sequence outlined in this chapter, helped to create a product-line architecture for CIAO and TAO that meets all these requirements, while still being compact and comprehensible. Further coverage on the patterns in TAO appears in [SC99].