Applying a Pattern Language to Develop Application-level Gateways

Douglas C. Schmidt
schmidt@uci.edu
http://www.ece.uci.edu/~schmidt/
Department of Electrical & Computer Science
University of California, Irvine 92607

This paper appeared as a chapter in the book *Design Patterns in Communications*, (Linda Rising, ed.), Cambridge University Press, 2000. An abridged version appeared in the journal *Theory and Practice of Object Systems*, special issue on Patterns and Pattern Languages, Wiley & Sons, Vol. 2, No. 1, December 1996..

Abstract

Developers of communication applications must address recurring design challenges related to efficiency, extensibility, and robustness. These challenges are often independent of application-specific requirements. Successful developers resolve these challenges by applying appropriate patterns and pattern languages. Traditionally, however, these patterns have been locked in the heads of expert developers or buried deep within complex system source code. The primary contribution of this paper is to describe a pattern language that underlies object-oriented communication software. In addition to describing each pattern in this language, the paper illustrates how knowledge of the relationships and trade-offs among the patterns helps guide the construction of reusable communication frameworks and applications.

1 Introduction

Communication software is the set of services and protocols that makes possible modern distributed systems and applications, such as web services, distributed objects, collaborative applications, and e-commerce systems [1]. Building, maintaining, and enhancing high-quality communication software is hard, however. Developers must have a deep understanding of many complex issues, such as service initialization and distribution, concurrency control, flow control, error handing, event loop integration, and fault tolerance. Successful communication applications created by experienced software developers must embody effective solutions to these issues.

It is non-trivial to separate the essence of successful communication software solutions from the details of particular implementations. Even when software is written using well-structured object-oriented (OO) frameworks and components, it can be hard to identify key roles and relationships. Moreover, operating system (OS) platform *features*, such as the absence or presence of multi-threading, or application *requirements*, such as best-effort vs. fault tolerance error handling, are often different. These differences can mask the underlying architectural commonality among software solutions for different applications in the same domain.

Capturing the core commonality of successful communication software is important for the following reasons:

1. *It preserves important design information for programmers who enhance and maintain existing software.* Often, this information only resides in the heads of the original developers. If this design information is not documented explicitly, therefore, it can be lost over time, which increases software entropy and decreases software maintainability and quality.

2. *It helps guide the design choices of developers who are building new communication systems.* By documenting common traps and pitfalls in their domain, patterns can help developers select suitable architectures, protocols, and platform features without wasting time and effort (re)implementing inefficient or error-prone solutions.

The goal of this paper is to demonstrate by example an effective way to capture and convey the essence of successful communication software by describing a *pattern language* used to build application-level *gateways*, which route messages between *peers* distributed throughout a communication system. Patterns represent successful solutions to problems that arise when building software [2]. When related patterns are woven together, they form a language that helps to

- Define a vocabulary for talking about software development problems; and
- Provide a process for the orderly resolution of these problems.

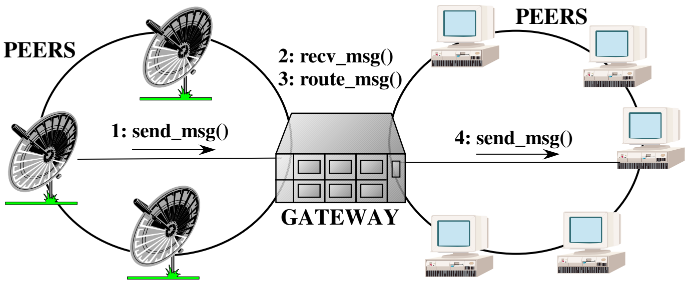

Figure 1: The Structure and Dynamics of Peers and an Application-level Gateway

Studying and applying patterns and pattern languages helps developers enhance the quality of their solutions by addressing fundamental challenges in communication software development. These challenges include communication of architectural knowledge among developers; accommodating new design paradigms or architectural styles; resolving non-functional forces, such as reusability, portability, and extensibility; and avoiding development traps and pitfalls that are usually learned only by costly trial and error [3].

This paper presents the OO architecture and design of an application-level gateway in terms of the pattern language used to guide the construction of reusable and gateway-specific frameworks and components. Application-level gateways have stringent requirements for reliability, performance, and extensibility. They are excellent exemplars, therefore, to present the structure, participants, and consequences of key patterns that appear in most communication software.

The pattern language described in this paper was discovered while building a wide range of communication systems, including on-line transaction processing systems, telecommunication switch management systems [4], electronic medical imaging systems [5], parallel communication subsystems [6], avionics mission computers [7], and real-time CORBA object request brokers (ORBs) [8]. Although the specific application requirements in these systems were different, the communication software design challenges were similar. This pattern language therefore embodies design expertise that can be reused broadly in the domain of communication software, well beyond the gateway example described in this paper.

The remainder of this paper is organized as follows: Section 1 outlines an OO software architecture for application-level gateways; Section 3 examines the patterns in the pattern language that forms the basis for reusable communication software, using application-level gateways as an example; Section 4 compares these patterns with other patterns in the literature; and Section 5 presents concluding remarks.

2 An OO Software Architecture for Application-level Gateways

A gateway is a mediator [2] that decouples cooperating peers throughout a network and allows them to interact without having direct dependencies on each other. As shown in Figure 1, messages routed through the gateway contain payloads encapsulated in routing messages. Figure 2 illustrates the structure, associations, and internal and external dynamics among objects within a software architecture for application-level gateways. This architecture is based on extensive experience developing gateways for various research and production communication systems. After building many gateway applications it became clear that their software architectures were largely independent of the protocols used to route messages to peers. This realization enabled the reuse of components depicted in Figure 2 for thousands of other communication software projects [1]. The ability to reuse these components so systematically stems from two factors:

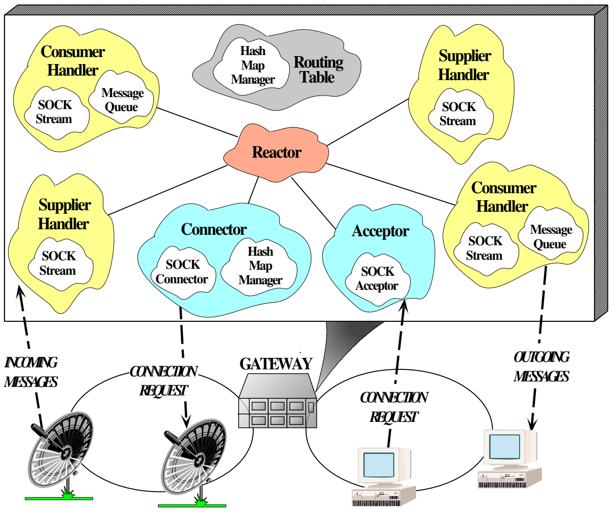

Figure 2: The OO gateway Software Architecture

1. *Understanding the actions and interactions of key patterns within the domain of communication software.* Patterns capture the structure and dynamics of participants in a software architecture at a higher level than source code and OO design models that focus on individual objects and classes. Some of the communication software patterns described in this paper have been documented individually [1]. Although individual pattern descriptions capture valuable design expertise, complex communication software systems embody scores of patterns. Understanding the relationships among these patterns is essential to document, motivate, and resolve difficult challenges that arise when building communication software. Therefore, Section 3 describes the interactions and relationships among these patterns in terms of a *pattern language* for communication software. The patterns in this language work together to solve complex problems within the domain of communication software.

2. *Developing an OO framework that implements these patterns.* Recognizing the patterns that commonly occur in many communication software systems helped shape the development of reusable framework components. The gateway systems this paper is based upon were implemented using the ADAPTIVE Communication Environment (ACE) framework [9], which provides integrated reusable C++ wrapper facades and components that perform common communication software tasks. These tasks include event demultiplexing, event handler dispatching, connection establishment, routing, dynamic configuration of application services, and concurrency control. In addition, the ACE framework contains implementations of the patterns described in Section 3. The patterns are much richer than their implementation in ACE, however, and have been applied by many other communication systems, as well.

This section describes how various ACE components have been reused and extended to implement the application-independent and application-specific components in the communication gateway shown in Figure 2. Following this overview, Section 3 examines the pattern language that underly the ACE components.

2.1 Application-independent Components

Most components in Figure 2 are based on ACE components that can be reused in other communication systems. The only components that are not widely reusable are the `Supplier` and `Consumer Handlers`, which implement the application-specific details related to message formats and the gateway's routing protocol. The behavior of the application-independent components in the gateway is outlined below:

Interprocess communication (IPC) components: The `SOCK Stream`, `SOCK Connector`, and `SOCK Acceptor` components encapsulate the socket network programming interface [9]. These components are implemented using the *Wrapper Facade* pattern [1], which simplifies the development of portable communication software by shielding developers from low-level, tedious, and error-prone socket-level programming. In addition, they form the foundation for higher-level patterns and ACE components described below.

Event demultiplexing components: The `Reactor` is an OO event demultiplexing mechanism based on the *Reactor* pattern [1] described in Section 3.3. It channels all external stimuli in an event-driven application through a single demultiplexing point. This design permits single-threaded applications to wait on event handles, demultiplex events, and dispatch event handlers efficiently. Events indicate that something significant has occurred, *e.g.,* the arrival of a new connection or work request. The main source of events in the gateway are (1) connection events that indicate requests to establish connections and (2) data events that indicate routing messages encapsulating various payloads, such as commands, status messages, and bulk data transmissions.

Initialization and event handling components: Establishing connections between endpoints involves two roles: (1) the *passive role*, which initializes an endpoint of communication at a particular address and waits passively for the other endpoint to connect with it and (2) the *active role*, which actively initiates a connection to one or more endpoints that are playing the passive role. The `Connector` and `Acceptor` are factories [2] that implement active and passive roles for initializing network services, respectively. These components implement the *Acceptor-Connector* pattern, which is described in Section 3.5. The gateway uses these components to establish connections with peers and produce initialized `Supplier` and `Consumer Handlers`.

To increase system flexibility, connections can be established in the following two ways:

1. *From the gateway to the peers*, which is often done to establish the initial system configuration of peerswhen the gateway first starts up.

2. *From a peer to the gateway*, which is often done after the system is running whenever a new peer wants to send or receive routing messages.

In a large system, dozens or hundreds of peers may be connected to a single gateway. To expedite initialization, therefore, the gateway's `Connector` can initiate all connections asynchronously rather than synchronously. Asynchrony helps decrease connection latency over long delay paths, such as wide-area networks (WANs) built over satellites or long-haul terrestrial links. The underlying `SOCK Connector` [9] contained within a `Connector` provides the low-level asynchronous connection mechanism. When a `SOCK Connector` connects two socket endpoints via TCP it produces a `SOCK Stream` object, which is then used to exchange data between that peer and the gateway.

Message demultiplexing components: The `Map Manager` efficiently maps external ids, such as peer routing addresses, onto internal ids, such as `Consumer Handlers`. The gateway uses a `Map Manager` to implement a `Routing Table` that handles the demultiplexing and routing of messages internally to a gateway. The `Routing Table` maps addressing information contained in routing messages sent by peers to the appropriate set of `Consumer Handlers`.

Message queueing components: The `Message Queue` [9] provides a generic queueing mechanism used by the gateway to buffer messages in `Consumer Handlers` while they are waiting to be routed to peers. A `Message Queue` can be configured to run efficiently and robustly in single-threaded or multi-threaded environments. When a queue is instantiated, developers can select the desired concurrency strategy. In multi-threaded environments, `Message Queues` are implemented using the *Monitor Object* pattern [1].

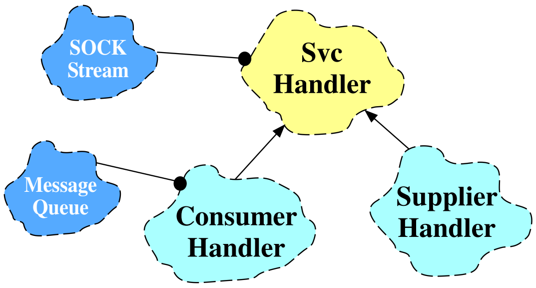

Figure 3: Supplier and Consumer Handler Inheritance Hierarchy

2.2 Application-specific Components

In Figure 2 only two of the components–`Supplier` and `Consumer Handlers`–are specific to the gateway application. These components implement the Non-blocking Buffered I/O pattern described in Section 3.6. `Supplier` and `Consumer Handlers` reside in the gateway, where they serve as proxies for the original source and the intended destination(s) of routing messages sent to hosts across the network. The behavior of these two gateway-specific components is outlined below:

Supplier Handlers: `Supplier Handlers` are responsible for routing incoming messages to their destination(s). The `Reactor` notifies a `Supplier Handler` when it detects an event on that connection's communication endpoint. After the `Supplier Handler` has received a complete routing message from that endpoint it consults the `Routing Table` to determine the set of `Consumer Handler` destinations for the message. It then requests the selected `Consumer Handler`(s) to forward the message to the appropriate peer destinations.

Consumer Handlers: A `Consumer Handler` is responsible for delivering routing messages to their destinations reliably. It implements a flow control mechanism to buffer bursts of routing messages that cannot be sent immediately due to transient network congestion or lack of buffer space at a receiver. Flow control is a transport layer mechanism that ensures a source peer does not send data faster than a destination peer can buffer and process the data. For instance, if a destination peer runs out of buffer space, the underlying TCP protocol instructs the associated gateway's `Consumer Handler` to stop sending messages until the destination peer consumes its existing data.

A gateway integrates the application-specific and application-independent components by customizing, instantiating, and composing the ACE components described above. As shown in Figure 3 `Supplier` and `Consumer Handlers` inherit from a common ancestor: the ACE `Svc Handler` class, which is produced by `Connectors` and `Acceptors`. Each `Svc Handler` is a local Proxy [2] for a remotely connected peer. It contains a `SOCK Stream`, which enables peers to exchange messages via connected socket handles.

A `Consumer Handler` is implemented in accordance with the Non-blocking Buffered I/O pattern. Thus, it uses an ACE `Message Queue` to chain unsent messages in the order they must be delivered when flow control mechanisms permit. After a flow controlled connection opens up, the ACE framework notifies its `Consumer Handler`, which starts draining the `Message Queue` by sending messages to the peer. If flow control occurs again this sequence of steps is repeated until all messages are delivered.

To improve reliability and performance, the gateways described in this paper utilize the Transmission Control Protocol (TCP). TCP provides a reliable, in-order, non-duplicated bytestream service for application-level gateways. Although TCP connections are inherently bi-directional, data sent from peer to the gateway use a different connection than data sent from the gateway to the peer. There are several advantages to separating input connections from output connections in this manner:

- It simplifies the construction of gateway `Routing Tables`;
- It allows more flexible connection configuration and concurrency strategies;
- It enhances reliability since `Supplier` and `Consumer Handlers` can be reconnected independently if errors occur on a connection.

3 A Pattern Language for Application-level Gateways

Section 1 described the structure and functionality of an application-level gateway. Although this architectural overview helps to clarify the behavior of key components in a gateway, it does not reveal the deeper relationships and roles that underly these software components. In particular, the architecture descriptions do not motivate *why* a gateway is designed in this particular manner or why certain components act and interact in certain ways. Understanding these relationships and roles is crucial to develop, maintain, and enhance communication software.

An effective way to capture and articulate these relationships and roles is to describe the *pattern language* that generates them. Studying the pattern language that underlies the gateway software provides the following two benefits:

1. *Identify successful solutions to common design challenges.* The pattern language underlying the gateway architecture transcends the particular details of the application and resolves common challenges faced by communication software developers. A thorough understanding of this pattern language enables widespread reuse of gateway software architecture in other systems, even when reuse of its algorithms, implementations, interfaces, or detailed designs is not feasible [10].

2. *Reduce the effort of maintaining and enhancing gateway software.* A pattern language helps to capture and motivate the collaboration between multiple classes and objects. This is important for developers who must maintain and enhance a gateway. Although the roles and relationships in a gateway design are embodied in the source code, extracting them from the surrounding implementation details can be costly and error-prone.

3.1 Strategic Patterns

Figure 4 illustrates the following five *strategic* patterns that form a portion of the language that generates connection-oriented, application-level gateways:

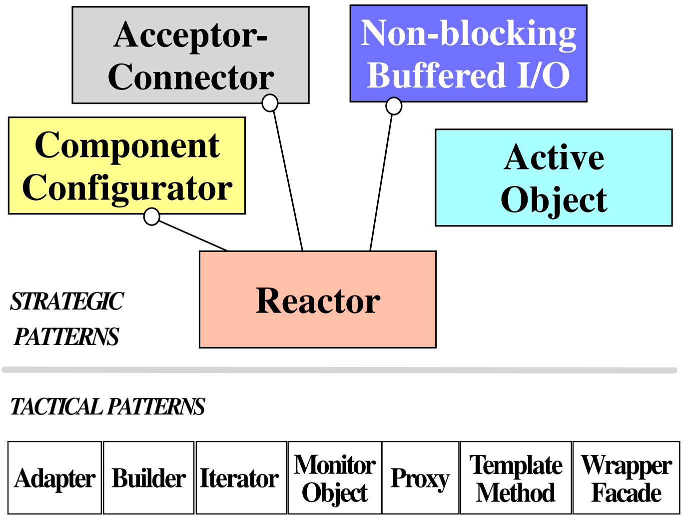

Figure 4: A Pattern Language for Application-level gateways

• **Reactor [1]:** This pattern structures event-driven applications, particularly servers, that receive requests from multiple clients concurrently but process them iteratively.

• **Active Object [1]:** This pattern decouples method execution from method invocation to enhance concurrency and simplify synchronized access to objects that reside in their own threads of control.

• **Component Configurator [1]:** This pattern allows an application to link and unlink its component implementations at run-time without having to modify, recompile or statically relink the application. It also supports the reconfiguration of components into different processes without having to shut down and re-start running processes.

• **Acceptor-Connector [1]:** This pattern decouples connection establishment and service initialization from service processing in a networked system.

• **The Non-blocking Buffered I/O pattern:** This pattern decouples input mechanisms and output mechanisms so that data can be routed correctly and reliably without blocking application processing unduly.

The five patterns in this language are strategic because they significantly influence the software architecture for applications in a particular domain, which in this case is the domain of communication software and gateways. For example, the Non-blocking Buffered I/O pattern described in Section 3.6 ensures that message processing is not disrupted or postponed indefinitely when a gateway experiences congestion or failure. This pattern helps to sustain a consistent quality-of-service (QoS) for gateways that use reliable connection-oriented transport protocols, such as TCP/IP or IPX/SPX. A thorough understanding of the strategic communication patterns described in this paper is essential to develop robust, efficient, and extensible communication software, such as application-level gateways.

3.2 Tactical Patterns

The application-level gateway implementation also uses many *tactical* patterns, such as the following:

• **Adapter [2]:** This pattern transforms a non-conforming interface into one that can be used by a client. The gateway uses this pattern to treat different types of routing messages, such as commands, status information, and bulk data, uniformly.

• **Builder [2]:** This pattern provides a factory for building complex objects incrementally. The gateway uses this pattern to create its `Routing Table` from a configuration file.

• **Iterator [2]:** This pattern decouples sequential access to a container from the representation of the container. The gateway uses this pattern to connect and initialize multiple `Supplier` and `Consumer Handlers` with their peers.

• **Monitor Object [1]:** This pattern synchronizes concurrent method execution to ensure that only one method at a time runs within an object. It also allows an object's methods to schedule their execution sequences co-operatively. The gateway uses this pattern to synchronize the multi-threaded configuration of its `Message Queues`.

• **Proxy [2]:** This pattern provides a local surrogate object that acts in place of a remote object. The gateway uses this pattern to shield the main gateway routing code from delays or errors caused by the fact that peers are located on other host machines in the network.

• **Template Method [2]:** This pattern specifies an algorithm where some steps are supplied by a derived class. The gateway uses this pattern to selectively override certain steps in its `Connector` and `Acceptor` components that that failed connections can be restarted automatically.

• **Wrapper Facade [1]:** This pattern encapsulates the functions and data provided by existing non-OO APIs within more concise, robust, portable, maintainable, and cohesive OO class interfaces. The ACE framework uses this pattern to provide an OS-independent set of concurrent network programming components used by the gateway.

Compared to the five strategic patterns outlined above, which are domain-specific and have broad design implications, these tactical patterns are domain-independent and have a relatively localized impact on a software design. For instance, Iterator is a tactical pattern used in the gateway to process entries in the `Routing Table` sequentially without violating data encapsulation. Although this pattern is domain-independent and thus widely applicable, the problem it addresses does not impact the application-level gateway software design as pervasively as strategic patterns, such as Non-blocking Buffered I/O or Reactor. A thorough understanding of tactical patterns is essential, however, to implement highly flexible software that is resilient to changes in application requirements and platform environments.

The remainder of this section describes each of the strategic patterns in detail and explains how they are used in the gateway.

3.3 The Reactor Pattern

Intent: The Reactor pattern structures event-driven applications, particularly servers, that receive requests from multiple clients concurrently but process them iteratively.

Motivation and forces: Single-threaded applications must handle events from multiple sources without blocking indefinitely on any particular source. The following forces impact the design of single-threaded, event-driven communication software:

1. *The need to demultiplex multiple types of events from multiple sources of events efficiently within a single thread of control.* Often, events from multiple sources within an application process must be handled at the event demultiplexing level. By handling events at this level, there may be no need for more complicated threading, synchronization, or locking within an application.

2. *The need to extend application behavior without requiring changes to the event dispatching framework.* Demultiplexing and dispatching mechanisms are often application-independent and can therefore be reused. In contrast, the event handler policies are more application-specific. By separating these concerns, application policies can change without affecting lower-level framework mechanisms.

Solution: Apply the Reactor pattern to wait synchronously for the arrival of indication events on one or more event sources such as connected socket handles. Integrate the mechanisms that demultiplex and dispatch the events to services that process them. Decouple these event demultiplexing and dispatching mechanisms from the application-specific processing of indication events within the services.

Structure, participants, and implementation: Figure 5 illustrates the structure and participants in the Reactor pattern. The `Reactor` defines an interface for registering, removing, and dispatching concrete event handler objects, such as `Supplier` or `Consumer Handlers` in the gateway. An

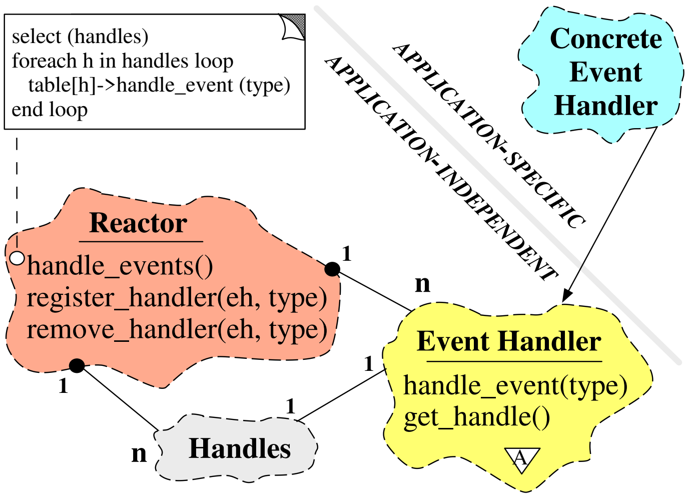

Figure 5: Structure and Participants in the Reactor Pattern

implementation of this interface provides a set of application-independent mechanisms. These mechanisms perform event demultiplexing and dispatching of application-specific event handlers in response to various types of input, output, and timer events.

An `Event Handler` specifies an abstract interface used by a `Reactor` to dispatch callback methods defined by objects that register to events of interest. A concrete event handler is a class that inherits from `Event Handler` and selectively overrides callback method(s) to process events in an application-specific manner.

Dynamics: Figure 6 illustrates the dynamics among participants in the Reactor pattern. These dynamics can be divided

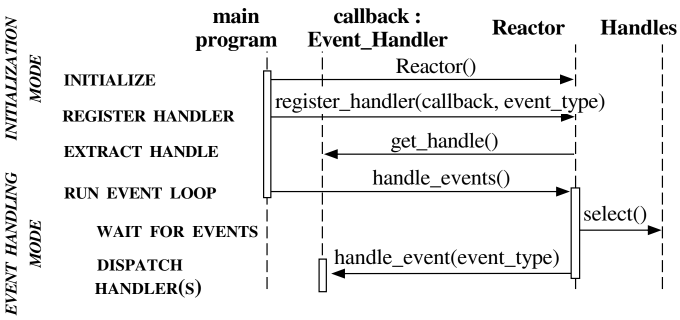

Figure 6: Dynamics for the Reactor Pattern

into the following two modes:

1. *Initialization mode*, where `Concrete Event Handler` objects are registered with the `Reactor`;

2. *Event handling mode*, where the `Reactor` invokes up-calls on registered objects, which then handle events in an application-specific way.

Usage: The `Reactor` is used for the following types of event dispatching operations in a gateway:

1. *Input events.* The `Reactor` dispatches each incoming routing message to the `Supplier Handler` associated with its socket handle, at which point the message is routed to the appropriate `Consumer Handler`(s). This use-case is shown in Figure 7.

2. *Output events.* The `Reactor` ensures that outgoing routing messages are reliably delivered over flow controlled `Consumer Handlers`, as described in Section 3.6 and 3.7.

3. *Connection completion events.* The `Reactor` dispatches events that indicate the completion status of connections that are initiated asynchronously. These events are used by the `Connector` component described in Section 3.5.

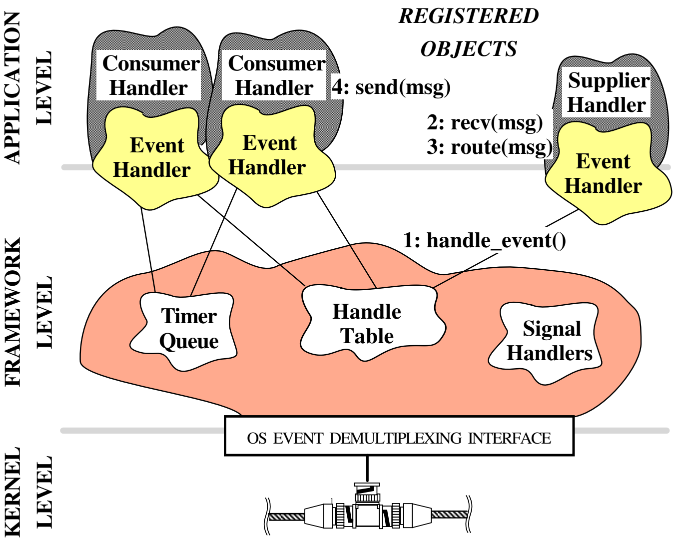

Figure 7: Using the Reactor Pattern in the gateway

4. *Connection request events.* The `Reactor` also dispatches events that indicate the arrival of passively initiated connections. These events are used by the `Acceptor` component described in Section 3.5.

The Reactor pattern has been used in many single-threaded event-driven frameworks, such as the Motif, Interviews [11], System V STREAMS [12], the ACE OO communication framework [9], and implementations of CORBA [8]. In addition, it provides the event demultiplexing infrastructure for all of the other strategic patterns presented below.

3.4 The Component Configurator Pattern

Intent: The Component Configurator pattern allows an application to link and unlink its component implementations at run-time without having to modify, recompile or statically re-link the application. It also supports the reconfiguration of components into different processes without having to shut down and re-start running processes.

Motivation and forces: The following forces impact the design of highly flexible and extensible communication software:

1. *The need to defer the selection of a particular implementation of a component until very late in the design cycle.* Deferring these configuration decisions until installation-time or run-time significantly increases the design choices available to developers. For example, run-time context information can be used to guide implementation decisions and components can be (re)configured into applications dynamically.

2. *The need to build complete applications by composing or scripting multiple independently developed components.*

Much of the recurring component configuration and initialization behavior of applications should be factored out into reusable methods. This separation of concerns allows new versions of components to be linked into an application at run-time without disrupting currently executing components.

Solution: Apply the Component Configurator pattern to decouple component interfaces from their implementations and make applications independent of the point(s) in time at which component implementations are configured into application processes.

Structure, participants, and implementation: Figure 8 illustrates the structure and participants of the Component Configurator pattern. This pattern reuses the Reactor pattern's

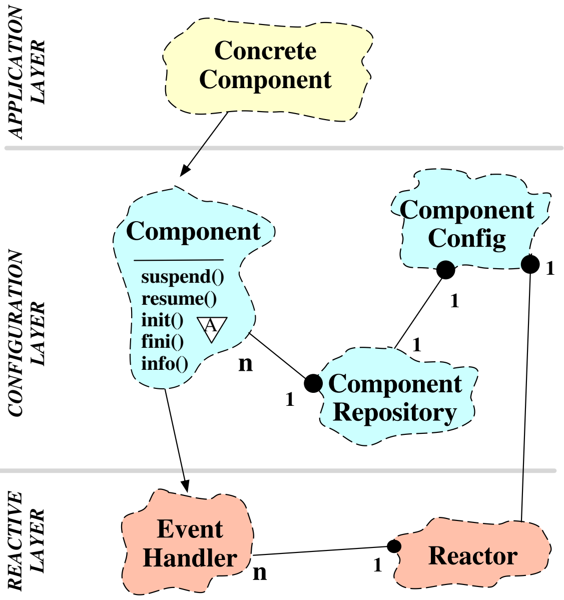

Figure 8: Structure and Participants in the Component Configurator Pattern

`Reactor` and `Event Handler` for its event demultiplexing and dispatching needs. The `Component` is a subclass of `Event Handler` that adds interfaces for initializing and terminating C++ objects when they are linked and unlinked dynamically. Application-specific components inherit from `Component` and selectively override its `init` and `fini` methods to implement custom initialization and termination behavior, respectively.

The `Component Repository` records which `Components` are currently linked and active. The `Component Config` is a facade [2] that orchestrates the behavior of the other components. It also provides a single access point for linking, activating, suspending, resuming, and unlinking `Components` into and out of an application at run-time.

Dynamics: Figure 9 illustrates the dynamics between participants in the Component Configurator pattern. These dynam-

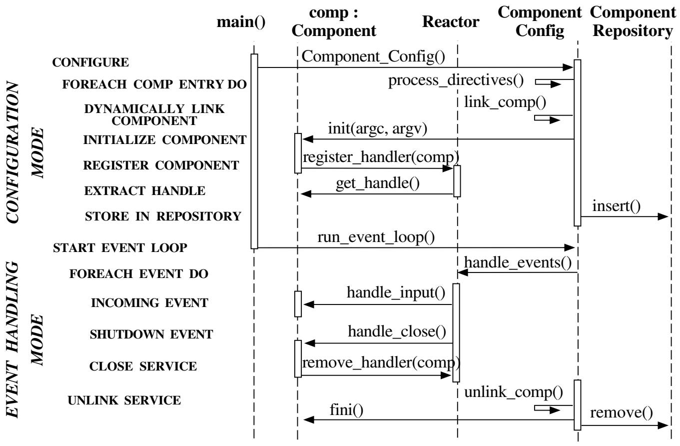

Figure 9: Dynamics for the Component Configurator Pattern

ics can be divided into the following two modes:

1. *Configuration mode*, which dynamically links or unlinks `Components` to and from an application.

2. *Event handling mode*, which process incoming events using patterns such as Reactor or Active Object [1].

Usage: The Component Configurator pattern is used in the gateway as shown in Figure 10. The `Reactive Gateway` component is a single-threaded implementation of the gateway that can be dynamically linked via commands in a configuration script. To dynamically replace this component with a multi-threaded implementation, the `Component Config` need only reconsult its `comp.conf` file, unlink the `Reactive Gateway`, dynamically link the `Thread-per Connection Gateway` or `Thread Pool Gateway`, and initialize the new implementation. The `Component Config` facade uses dynamic linking to implement the Component Configurator pattern efficiently.

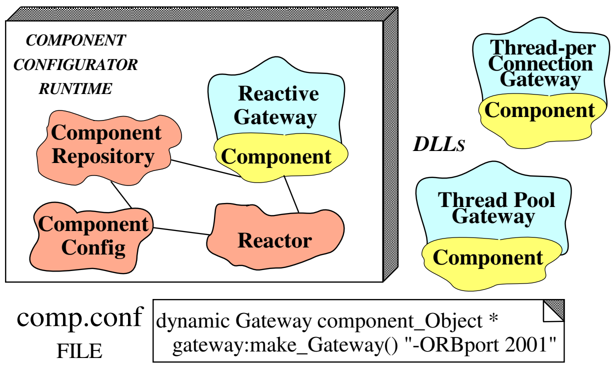

Figure 10: Using the Component Configurator Pattern in the gateway

The Component Configurator pattern is used in the Windows NT Service Control Manager (SCM), which allows a master SCM process to initiate and control administrator-installed service components automatically. In general, modern operating systems, such as Solaris, Linux, and Windows NT, provide support for dynamically-configured kernel-level device drivers that implement the Component Configurator pattern. Another use of the Component Configurator pattern is the applet mechanism in Java, which supports dynamic downloading, initializing, starting, stopping, and terminating of Java applets.

3.5 The Acceptor-Connector Pattern

Intent: The Acceptor-Connector pattern decouples connection establishment and service initialization from service processing in a networked system.

Motivation and forces: Connection-oriented applications, such as our application-level gateway, and middleware, such as CORBA, are often written using lower-level network programming interfaces, like sockets [13]. The following forces impact the initialization of services written using these lower-level interfaces:

1. *The need to reuse connection establishment code for each new service.* Key characteristics of services, such as the communication protocol or the data format, should be able to evolve independently and transparently from the mechanisms used to establish the connections. Since service characteristics change more frequently than connection establishment mechanisms, separating these concerns helps to reduce software coupling and increase code reuse.

2. *The need to make the connection establishment code portable across platforms that contain different network programming interfaces.* Parameterizing the Acceptor-Connector's mechanisms for accepting connections and performing services helps to improve portability by allowing the wholesale replacement of these mechanisms. This makes the connection establishment code portable across platforms that contain different network programming interfaces, such as sockets but not TLI, or vice versa.

3. *The need to enable flexible service concurrency policies.* After a connection is established, peer applications use the connection to exchange data to perform some type of service, such as remote login or HTML document transfer. A service can run in a single-thread, in multiple threads, or multiple processes, regardless regardless of how the connection was established or how the services were initialized.

4. *The need to ensure that a passive-mode I/O handle is not accidentally used to read or write data.* By strongly decoupling the connection establishment logic from the service processing logic, passive-mode socket endpoints cannot be used incorrectly, *e.g.,* by trying to read or write data on a passive-mode listener socket used to accept connections. This eliminates an important class of network programming errors.

5. *The need to actively establish connections with large number of peers efficiently.* When an application must establish connections with a large number of peers efficiently over long-delay WANs it may be necessary to use asynchrony and initiate and complete multiple connections in non-blocking mode.

Solution: Apply the Acceptor-Connector pattern to decouple the connection and initialization of peer services in a networked application from the processing these peer services perform after they are connected and initialized.

Structure, participants, and implementation: Figure 11 illustrates the layering structure of participants in the Acceptor-Connector pattern. The `Acceptor` and `Connector` components are factories that assemble the resources necessary to connect and activate `Svc Handlers`. `Svc Handlers` are components that exchange messages with connected peers.

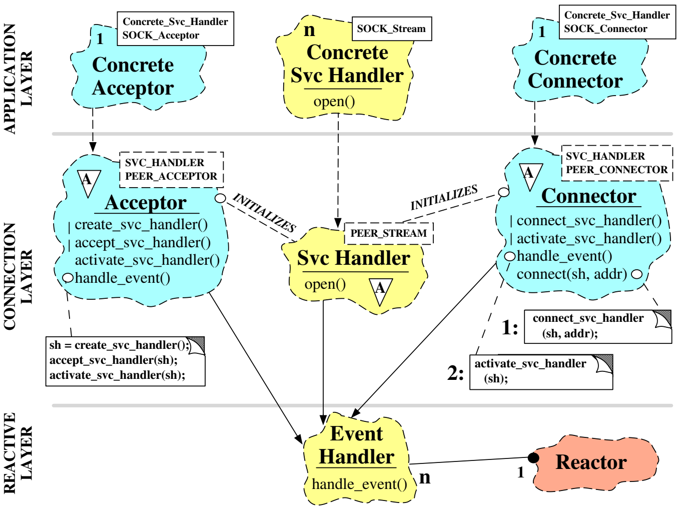

Figure 11: Structure and Participants in the Acceptor-Connector Pattern

The participants in the Connection Layer of the Acceptor-Connector pattern leverage off the Reactor pattern. For instance, the `Connector`'s asynchronous initialization strategy establishes a connection after the `Reactor` notifies it that a previously initiated connection request to a peer has completed. Using the Reactor pattern enables multiple `Svc Handlers` to be initialized asynchronously within a single thread of control.

To increase flexibility, `Acceptor` and `Connector` components can be parameterized by a particular type of IPC mechanism and SVC HANDLER. The IPC mechanism supplies the underlying transport mechanism, such as C++ wrapper facades for sockets or TLI, used to establish a connection. The SVC HANDLER specifies an abstract interface for defining a service that communicates with a connected peer. A Svc Handler can be parameterized by a PEER STREAM endpoint. The Acceptor and Connector components associate this endpoint to its peer when a connection is established.

By inheriting from Event Handler, a Svc Handler can register with a Reactor and use the Reactor pattern to handle its I/O events within the same thread of control as the Acceptor or Connector. Conversely, a Svc Handler can use the Active Object pattern and handle its I/O events in a separate thread. Section 3.7 evaluates the tradeoffs between these two patterns.

Parameterized types are used to decouple the Acceptor-Connector pattern's connection establishment strategy from the type of service and the type of connection mechanism. Developers supply template arguments for these types to produce Application Layer Acceptor or Connectors, such as the Connector used by the gateway to initialize its Supplier and Consumer Handlers. This design enables the wholesale replacement of the SVC HANDLER and IPC mechanism, without affecting the Acceptor-Connector pattern's service initialization strategy.

Note that a similar degree of decoupling could be achieved via inheritance and dynamic binding by using the Abstract Factory or Factory Method patterns described in [2]. Parameterized types were used to implement this pattern since they improve run-time efficiency. In general, templates trade compile- and link-time overhead and space overhead for improved run-time performance.

Dynamics: Figure 12 illustrates the dynamics among participants for the Acceptor component of the pattern. These dynamics are divided into the following three phases:

1. *Endpoint initialization phase*, which creates a passive-mode endpoint encapsulated by PEER ACCEPTOR that is bound to a network address, such as an IP address and port number. The passive-mode endpoint listens for connection requests from peers. This endpoint is registered with the Reactor, which drives the event loop that waits on the endpoint for connection requests to arrive from peers.

2. *Service activation phase.* Since an Acceptor inherits from an Event Handler the Reactor can dispatch the Acceptor's handle_event method when connection request events arrive. This method performs the Acceptor's Svc Handler initialization strategy, which (1) assembles the resources necessary to create a new Concrete Svc Handler object, (2) accepts the connection into this object,

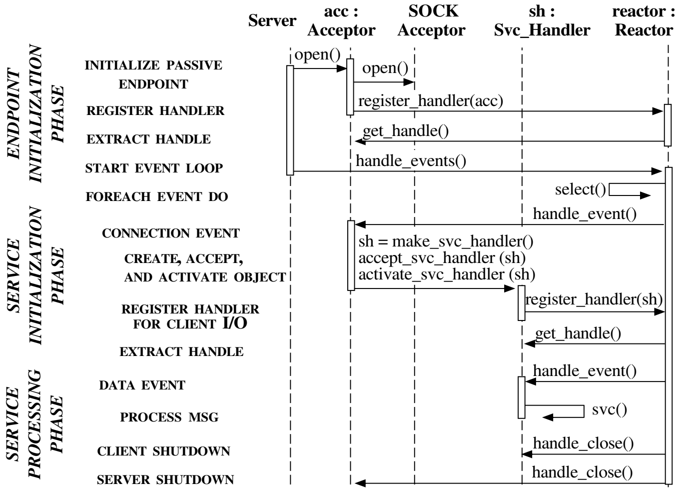

Figure 12: Dynamics for the Acceptor Component

and (3) activates the Svc Handler by calling its open hook method.

3. *Service processing phase.* After the Svc Handler is activated, it processes incoming event messages arriving on the PEER STREAM. A Svc Handler can process incoming event messages using patterns such as the Reactor or the Active Object [1].

The dynamics among participants in Connector component of the pattern can be divided into the following three phases:

1. *Connection initiation phase*, which actively connects one or more Svc Handlers with their peers. Connections can be initiated synchronously or asynchronously. The Connector's connect method implements the strategy for establishing connections actively.

2. *Service initialization phase*, which activates a Svc Handler by calling its open method when its connection completes successfully. The open method of the Svc Handler then performs service-specific initialization.

3. *Service processing phase*, which performs the application-specific service processing using the data exchanged between the Svc Handler and its connected peer.

Figure 13 illustrates these three phases of dynamics using *asynchronous* connection establishment. Note how the Connector's connection initiation phase is separated *temporally* from the service initialization phase. This design enables multiple connection initiations to proceed in parallel within a single thread of control. The dynamics for synchronous connection establishment is similar. In this case, the Connector

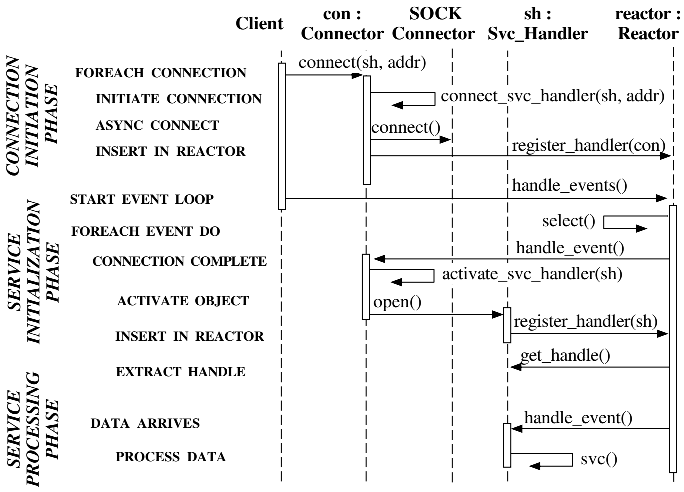

Figure 13: Dynamics for the Asynchronous `Connector` Component

combines the connection initiation and service initialization phases into a single blocking operation.

In general, synchronous connection establishment is useful for the following situations:

- If the latency for establishing a connection is very low, such as establishing a connection with a server on the same host via the loopback device.
- If multiple threads of control are available and it is feasible to use a different thread to connect each `Svc Handler` synchronously.
- If a client application cannot perform useful work until a connection is established.

In contrast, asynchronous connection establishment is useful for the following situations:

- If the connection latency is high and there are many peers to connect with, *e.g.*, establishing a large number of connections over a high-latency WAN.
- If only a single thread of control is available, *e.g.,* if the OS platform does not provide application-level threads.
- If the client application must perform additional work, such as refreshing a GUI, while the connection is in the process of being established.

It is often the case that network services, such as our application-level gateway, must be developed without knowing if they will connect synchronously or asynchronously. Therefore, components provided by a general-purpose network programming framework must support multiple synchronous and asynchronous use-cases.

The Acceptor-Connector pattern increases the flexibility and reuse of networking framework components by separating the connection establishment logic from the service processing logic. The only coupling between (1) `Acceptor` and `Connector` components and (2) a `Svc Handler` occurs in the service initialization phase, when the `open` method of the `Svc Handler` is invoked. At this point, the `Svc Handler` can perform its service-specific processing using any suitable application-level protocol or concurrency policy.

For instance, when messages arrive at a gateway, the `Reactor` can be used to dispatch `Supplier Handlers` to frame the messages, determine outgoing routes, and deliver the messages to their `Consumer Handlers`. However, `Consumer Handlers` can send the data to the remote destinations using a different type of concurrency mechanism, such as Active Objects described in Section 3.7.

Usage: Figure 14 illustrates how the `Acceptor` component of the Acceptor-Connector pattern is used by the gateway when it plays the passive connection role. In this case, peers

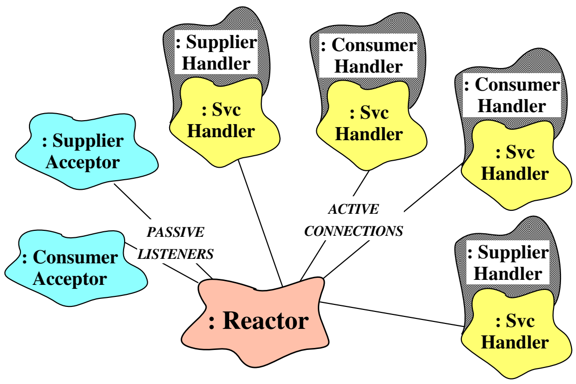

Figure 14: Using the `Acceptor` Component in the gateway

connect to gateway, which uses the `Acceptor` to decouple the passive initialization of `Supplier` and `Consumer Handlers` from the routing tasks performed after a handler is initialized.

Figure 15 illustrates how the `Connector` component of the Acceptor-Connector pattern is used by the gateway to simplify the task of connecting to a large number of peers. In this case, peer addresses are read from a configuration file during gateway initialization. The gateway uses the Builder pattern [2] to bind these addresses to dynamically allocated `Consumer Handlers` or `Supplier Handlers`. Since these handlers inherit from `Svc Handler`, all connections can be initiated asynchronously using the Iterator pattern [2]. The connections are then completed in parallel using the `Connector`.

Figure 15 shows the state of the `Connector` after four connections have been established. Three other connections that have not yet completed are owned by the `Connector`.

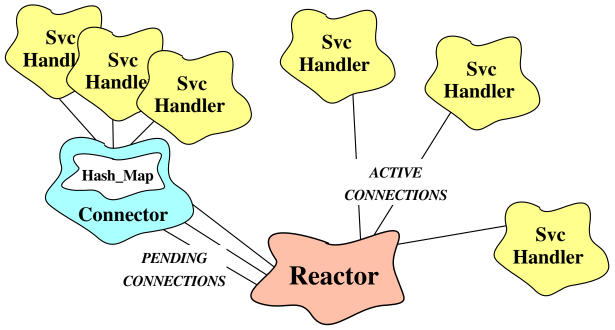

Figure 15: Using the `Connector` Component in the gateway

As shown in this figure, the `Connector` maintains a table of the three `Handlers` whose connections are pending completion. As connections complete, the `Connector` removes each connected `Channel` from its table and activates it. In the single-threaded implementation `Supplier Handlers` register themselves with the `Reactor` after they are activated. Henceforth, when routing messages arrive, `Supplier Handlers` receive and forward them to `Consumer Handlers`, which deliver the messages to their destinations (these activities are described in Section 3.6).

In addition to establishing connections, a gateway can use the `Connector` in conjunction with the `Reactor` to ensure that connections are restarted if network errors occur. This enhances the gateway's fault tolerance by ensuring that channels are automatically reinitiated when they disconnect unexpectedly, *e.g.,* if a peer crashes or an excessive amount of data is queued at a `Consumer Handler` due to network congestion. If a connection fails unexpectedly, an exponential-backoff algorithm can restart the connection efficiently by using the timer dispatching capabilities of the `Reactor`.

The intent and general architecture of the Acceptor-Connector pattern is found in network server management tools like `inetd` [13] and `listen` [14]. These tools utilize a master Acceptor process that listens for connections on a set of communication ports. Each port is associated with a communication-related service (such as the standard Internet services `ftp`, `telnet`, `daytime`, and `echo`). When a service request arrives on a monitored port, the Acceptor process accepts the request and dispatches an appropriate pre-registered handler that performs the service.

3.6 The Non-blocking Buffered I/O Pattern

Intent: The Non-blocking Buffered I/O pattern decouples input mechanisms and output mechanisms so that data can be routed correctly and reliably without blocking application processing unduly.

Motivation and forces: Message routing in a gateway must not be disrupted or postponed indefinitely when congestion or failure occurs on incoming and outgoing network connections. Thus, the following forces must be resolved when building robust connection-oriented gateways:

1. *The need to prevent misbehaving connections from disrupting the QoS of well-behaved connections.* Input connections can fail because peers disconnect. Likewise, output connections can flow control as a result of network congestion. In these types of cases, the gateway must not perform blocking `send` or `recv` operations on any single connection since (1) the entire gateway can hang indefinitely or (2) messages on other connections cannot be sent or received and the QoS provided to peers will degrade.

2. *The need to allow different concurrency strategies for processing input and output.* Several concurrency strategies can be used to process input and output, including (1) single-threaded processing using the Reactor pattern (Section 3.3) and (2) multi-threaded processing using the Active Object pattern (Section 3.7). Each strategy is appropriate under different situations, depending on factors such as the number of CPUs, context switching overhead, and number of peers.

Solution: Apply the Non-blocking Buffered I/O pattern to decoupling input processing from output processing to prevent blocking and allow customized concurrency strategies to be configured flexibly into an application.

Structure, participants, and implementation: Figure 16 illustrates the layer structuring of participants in the Non-blocking Buffered I/O pattern. The I/O Layer provides an

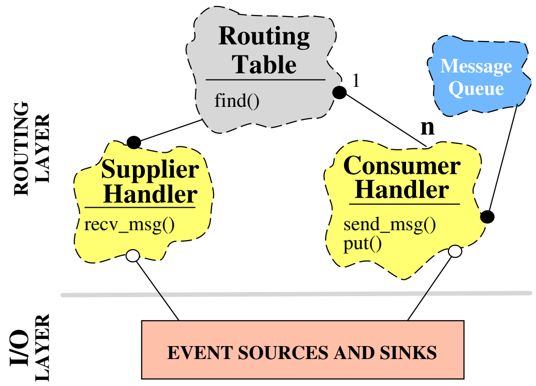

Figure 16: Structure and Participants in the Non-blocking Buffered I/O Pattern

event source for `Supplier Handlers` and an event sink for `Consumer Handlers`. A `Supplier Handler` uses a `Routing Table` to map routing messages onto one or more `Consumer Handlers`. If messages cannot be delivered to their destination peers immediately they are buffered in a `Message Queue` for subsequent transmission.

Since `Supplier Handlers` are decoupled from `Consumer Handlers` their implementations can vary independently. This separation of concerns is important since it allows the use of different concurrency strategies for input and output. The consequences of this decoupling is discussed further in Section 3.7.

Dynamics: Figure 17 illustrates the dynamics among participants in the Non-blocking Buffered I/O pattern. These dynamics can be divided into three phases:

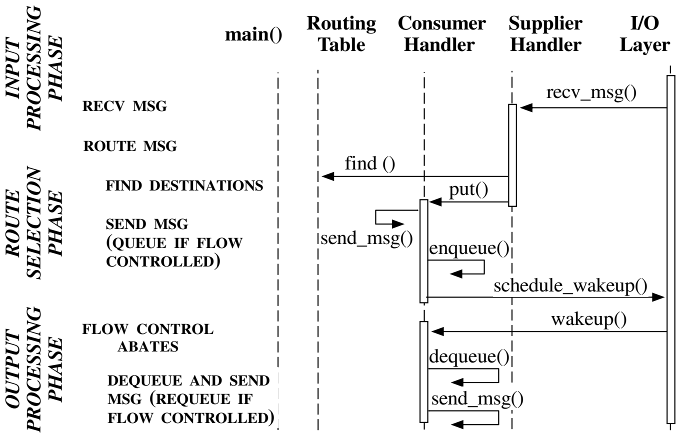

Figure 17: Dynamics for the Non-blocking Buffered I/O Pattern

1. *Input processing phase*, where `Supplier Handlers` reassemble incoming TCP segments into complete routing messages *without* blocking the application process.

2. *Route selection phase*. After a complete message has been reassembled, the `Supplier Handler` consults a `Routing Table` to select the `Consumer Handler(s)` responsible for sending the routing messages to their peer destinations.

3. *Output processing phase*, where the selected `Consumer Handlers` transmit the routing messages to their destination(s) *without* blocking the application process.

Usage: The other strategic patterns in this paper–*i.e.,* Reactor, Connector, Acceptor, and Active Object–can be applied to many types of communication software. In contrast, the Non-blocking Buffered I/O pattern is more coupled with gateway-style applications that route messages between peers. A primary challenge of building a reliable connection-oriented gateway centers on avoiding blocking I/O. This challenge centers primarily on reliably managing *flow control* that occurs on the connections used by `Consumer Handlers` to forward messages to peers. If the gateway blocked indefinitely when sending on a congested connection then incoming messages could not be routed, even if those messages were destined for non-flow controlled `Consumer Handlers`.

The remainder of Section 3.6 describes how the Non-blocking Buffered I/O pattern can be implemented in a single-threaded, reactive version of the gateway (Section 3.7 examines the multi-threaded, Active Object version of the Non-blocking Buffered I/O pattern). In this implementation, the Non-blocking Buffered I/O pattern uses a `Reactor` as a cooperative multi-tasking scheduler for gateway I/O operations on different connections within a single thread. Single-threading eliminates the following overhead:

- *Synchronization – e.g.*, access to shared objects like the `Routing Table` need not be serialized; and
- *Context switching – e.g.*, all message routing can occur within a single thread.

In the reactive implementation of the Non-blocking Buffered I/O pattern, the `Supplier Handlers` and `Consumer Handlers` are descendants of `Event Handler`. This layered inheritance design enables the gateway to route messages by having the `Reactor` dispatch the `handle_event` methods of `Supplier` and `Consumer Handlers` when messages arrive and flow control conditions subside, respectively.

Using the Reactor pattern to implement the Non-blocking Buffered I/O pattern involves the following steps:

1. *Initialize non-blocking endpoints.* The `Supplier` and `Consumer Handler` handles are set into non-blocking mode after they are activated by an `Acceptor` or `Connector`. The use of non-blocking I/O is essential to avoid blocking that can otherwise occur on congested network links.

2. *Input message reassembly and routing.* Routing messages are received in fragments by `Supplier Handlers`. If an entire message is not immediately available, the `Supplier Handler` must buffer the fragment and return control to the event loop. This is essential to prevent "head of line" blocking on `Supplier channels`. When a `Supplier Channel` successfully receives and frames an entire message it uses the `Routing Table` to determine the appropriate set of `Consumer Handlers` that will deliver the message.

3. *Message delivery.* The selected `Consumer Handlers` try to send the message to the destination peer. Messages must be delivered reliably in "first-in, first-out" (FIFO) order. To avoid blocking, all `send` operations in `Consumer Handlers` must check to make sure that the network link is not flow controlled. If it is *not*, the message can be sent successfully. This path is depicted by the `Consumer Handler` in the upper right-hand corner of Figure 18. If the link *is* flow controlled, however, the Non-blocking Buffered I/O pattern implementation must use

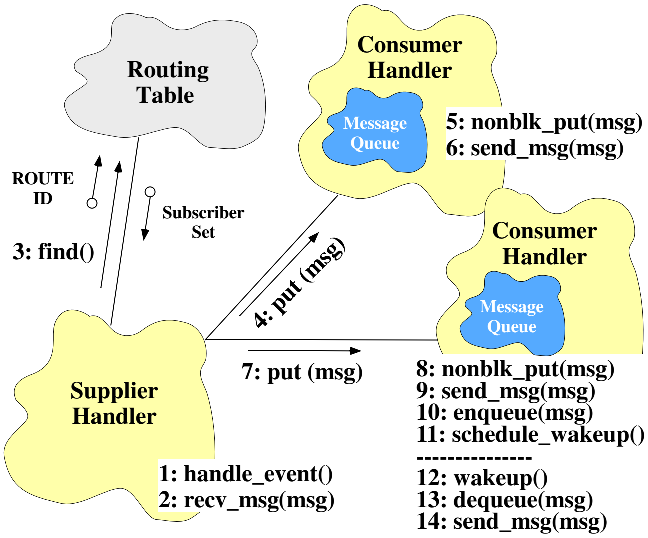

Figure 18: Using the Non-blocking Buffered I/O Pattern in a Single-threaded Reactive gateway

a different strategy. This path is depicted by the `Consumer Handler` in the lower right-hand corner of Figure 18.

To handle flow controlled connections, the `Consumer Handler` inserts the message it is trying to send into its `Message Queue`. It then instructs the `Reactor` to call back to the `Consumer Handler` when the flow control conditions abate, and returns to the main event loop. When it is possible to try to `send` again, the `Reactor` dispatches the `handle_event` method on the `Consumer Handler`, which then retries the operation. This sequence of steps may be repeated multiple times until the entire message is transmitted successfully.

Note that the gateway always returns control to its main event loop immediately after every I/O operation, regardless of whether it sent or received an entire message. This is the essence of the Non-blocking Buffered I/O pattern – it correctly routes the messages to peers without blocking on any single I/O channel.

3.7 The Active Object Pattern

Intent: The Active Object pattern decouples method execution from method invocation to enhance concurrency and simplify synchronized access to objects that reside in their own threads of control.

Motivation and forces: All the strategic patterns used by the single-threaded gateway in Section 3.6 are layered upon the Reactor pattern. The Acceptor-Connector and Non-blocking Buffered I/O patterns both use the Reactor as a scheduler/dispatcher to initialize and route messages within a single thread of control. In general, the Reactor pattern forms the central event loop in single-threaded reactive systems. For example, in the single-threaded gateway implementation, the `Reactor` provides a coarse-grained form of concurrency control that serializes the invocation of event handlers at the level of event demultiplexing and dispatching within a process. This eliminates the need for additional synchronization mechanisms within a gateway and minimizes context switching.

The Reactor pattern is well-suited for applications that use short-duration callbacks, such as passive connection establishment in the Acceptor pattern. It is less appropriate, however, for long-duration operations, such as blocking on flow controlled `Consumer Handlers` during periods of network congestion. In fact, much of the complexity in the single-threaded Non-blocking Buffered I/O pattern implementation stems from using the Reactor pattern as a cooperative multitasking mechanism. In general, this pattern does not adequately resolve the following force that impacts the design of applications, such as the gateway, that must communicate simultaneously with multiple peers:

1. *The need to ensure that blocking read and write operations on one endpoint do not detract from the QoS of other endpoints.* Network services are often easier to program if blocking I/O is used rather than reactive non-blocking I/O [1]. The simplicity occurs because execution state can be localized in the activation records of a thread, rather than be decentralized in a set of control blocks maintained explicitly by application developers.

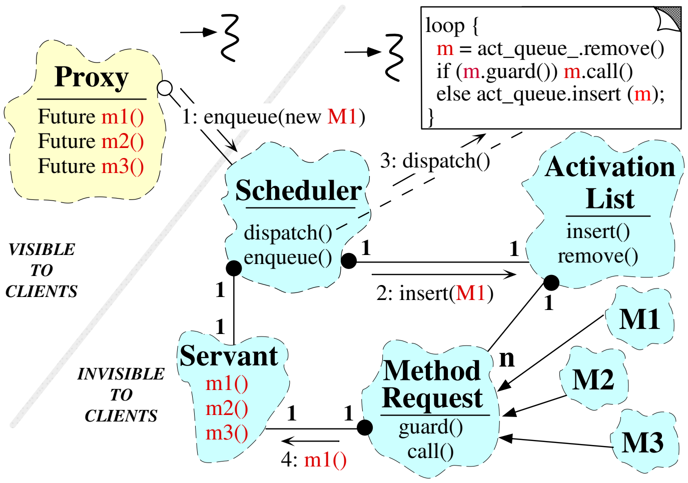

Figure 19: Structure and Participants in the Active Object Pattern

Solution: Apply the Active Object pattern to decouple method invocation on an object from method execution. Method invocation should occur in the client's thread of control, whereas method execution should occur in a separate thread. Moreover, design the decoupling so the client thread appears to invoke an ordinary method.

Structure, participants, and implementation: Figure 19 illustrates the structure and participants in the Active Object pattern. The `Proxy` exports the active object's public methods to clients. The `Scheduler` determines the next method to execute based on synchronization and scheduling constraints. The `Activation List` maintains a queue of pending `Method Requests`. The `Scheduler` determines the order in which these `Method Requests` are executed (a FIFO scheduler is used in the gateway to maintain the order of message delivery). The `Servant` maintains object state shared by the implementation methods.

Dynamics: Figure 20 illustrates the dynamics among participants in the Active Object pattern. These dynamics are divided into the following three phases:

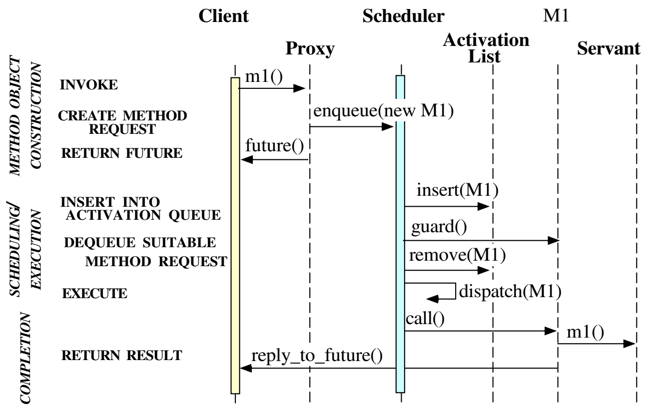

Figure 20: Dynamics for the Active Object Pattern

1. *Method Request construction.* In this phase, the client application invokes a method defined by the `Proxy`. This triggers the creation of a `Method Request`, which maintains the argument bindings to the method, as well as any other bindings required to execute the method and return a result. A binding to a `Future` object is returned to the caller of the method.

2. *Scheduling/execution.* In this phase the `Scheduler` acquires a mutual exclusion lock, consults the `Activation Queue` to determine which `Method Requests(s)` meet the synchronization constraints. The `Method Request` is then bound to the current `Servant` and the method is allowed to access/update the `Servant`'s state.

3. *Return result.* The final phase binds the result of the `Method Request` to a `Future` [15], which passes return values back to the caller when the method finishes executing. A `Future` is a synchronization object that enforces "write-once, read-many" synchronization. Subsequently, any readers that rendezvous with the `Future` will evaluate the future and obtain the result value. The `Future` and the `Method Request` can be garbage collected when they are no longer needed.

Usage: The gateway implementation described in Section 3.6 is single-threaded. It uses the Reactor pattern implementation of the Non-blocking Buffered I/O Pattern as a cooperative multi-tasking scheduler that dispatches events of interest to a gateway. After implementing a number of single-threaded gateways it became clear that using the Reactor pattern as the basis for all gateway routing I/O operations was error-prone and hard to maintain. For example, it was hard to remember why control must be returned promptly to the `Reactor`'s event loop when I/O operations cannot proceed. This misunderstanding became a common source of errors in single-threaded gateways.

To avoid these problems, a number of multi-threaded gateways were built using variations of the Active Object pattern. This pattern allows `Consumer Handlers` to block independently when sending messages to peers. The remainder of this section describes how `Consumer Handlers` can be multi-threading using the Active Object pattern.[1] This modification simplified the implementation of the Non-blocking Buffered I/O pattern substantially since `Consumer Handlers` can block in their own active object thread without affecting other `Handlers`. Implementing the `Consumer Handlers` as active objects also eliminated the subtle and error-prone cooperative multi-tasking programming techniques required when using the `Reactor` to schedule `Consumer Handlers`.

Figure 21 illustrates the Active Object version of the Non-blocking Buffered I/O pattern. Note how much simpler is it compared with the Reactor solution in Figure 18. This simplification occurs since the complex output scheduling logic is moved into the Active Objects, rather than being the responsibility of application developers.

It is also possible to observe the difference in complexity between the single-threaded and multi-threaded gateways by examining the source code that implements the Non-blocking Buffered I/O pattern in production gateway systems. It is hard to identify the reasons for this complexity simply by inspecting the source code due to all the error handling and protocol-specific details surrounding the implementation. These details tend to disguise the key insight: *the main difference between the complexity of the single-threaded and multi-threaded solutions arise from the choice of the Reactor pattern rather than*

[1] While it is possible to apply the Active Object pattern to the `Supplier Handlers` this has less impact on the gateway design because the `Reactor` already supports non-blocking input.

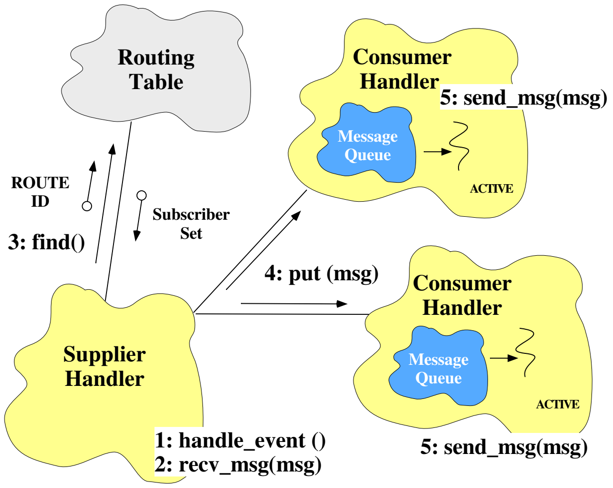

Figure 21: Using the Non-blocking Buffered I/O Pattern in a Multi-threaded Active Object gateway

the Active Object pattern.

This paper has explicitly focused on the interactions and tradeoffs between the Reactor and Active Object patterns to clarify the consequences of different design choices. In general, documenting the interactions and relationships between closely related patterns is a challenging and unresolved topic that is being addressed by the patterns community.

4 Related Patterns

[2, 16, 1] identify, name, and catalog many fundamental architectural and design patterns. This section examines how the patterns described in this paper relate to other patterns in this literature. Note that many of the tactical patterns outlined in Section 3.2 form the basis for implementing the strategic patterns presented in this paper.

The Reactor pattern is related to the Observer pattern [2]. In the Observer pattern, *multiple* dependents are updated automatically when a subject changes. In the Reactor pattern, a handler is dispatched automatically when an event occurs. Thus, the Reactor dispatches a *single* handler for each event, although there can be multiple sources of events. The Reactor pattern also provides a Facade [2]. The Facade pattern presents an interface that shields applications from complex relationships within a subsystem. The Reactor pattern shields applications from complex mechanisms that perform event demultiplexing and event handler dispatching.

The Component Configurator pattern is related to the Builder and Mediator patterns [2]. The Builder pattern provides a factory for constructing complex objects incrementally. The Mediator coordinates interactions between its associates. The Component Configurator pattern provides a factory for configuring and initializing components into an application at run-time. At run-time, the Component Configurator pattern allows the components offered by an application to be incrementally modified without disturbing executing components. In addition, the Component Configurator pattern coordinates the interaction between components configured into an application and external administrators that want to update, suspend, resume, or remove components at run-time.

The Acceptor-Connector pattern is related to the Template Method, Strategy, and Factory Method patterns [2]. In the Template Method pattern, an algorithm is written such that some steps are supplied by a derived class. In the Factory Method pattern, a method in a subclass creates an associate that performs a particular task, but the task is decoupled from the protocol used to create the task. The `Acceptor` and `Connector` components in the Acceptor-Connector pattern are factories that use template methods or strategies to create, connect, and activate handlers for communication channels. The intent of the Acceptor-Connector pattern is similar to the Client/Dispatcher/Server pattern described in [16]. They both are concerned with separating active connection establishment from the subsequent service. The primary difference is that the Acceptor-Connector pattern addresses both passive/active and synchronous/asynchronous connection establishment.

The Non-blocking Buffered I/O pattern is related to the Mediator pattern [2], which decouples cooperating components of a software system and allows them to interact without having direct dependencies among each other. The Non-blocking Buffered I/O pattern is specialized to resolve the forces associated with network communication. It decouples the mechanisms used to process input messages from the mechanisms used to process output mechanisms to prevent blocking. In addition, the Non-blocking Buffered I/O pattern allows the use of different concurrency strategies for input and output channels.

5 Concluding Remarks

This paper illustrates the application of pattern language that enables widespread reuse of design expertise and software components in production communication gateways. The patterns in this language illustrate the structure of, and collaboration between, objects that perform core communication software tasks. The tasks addressed by these patterns include event demultiplexing and event handler dispatching, connection establishment and initialization of application services, concurrency control, and routing.

The pattern language and ACE framework components described in this paper have been reused by the author and his colleagues in many production communication software systems ranging from telecommunication, electronic medical imaging, and avionics projects [10, 5, 7] to academic research projects [9, 8]. In general, this pattern language has aided the development of components and frameworks in these systems by capturing the structure and dynamics of participants in a software architecture at a level higher than (1) source code and (2) OO design models that focus on individual objects and classes.

An in-depth discussion of our experiences and lessons learned using patterns appeared in [4]. An ACE-based example of single-threaded and multi-threaded gateways that illustrates all the patterns in this paper is freely available at `www.cs.wustl.edu/~schmidt/ACE.html`.

References

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

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

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

[4] D. C. Schmidt, "Experience Using Design Patterns to Develop Reuseable Object-Oriented Communication Software," *Communications of the ACM (Special Issue on Object-Oriented Experiences)*, vol. 38, October 1995.

[5] I. Pyarali, T. H. Harrison, and D. C. Schmidt, "Design and Performance of an Object-Oriented Framework for High-Performance Electronic Medical Imaging," in *Proceedings of the* 2^{nd} *Conference on Object-Oriented Technologies and Systems*, (Toronto, Canada), USENIX, June 1996.

[6] D. C. Schmidt and T. Suda, "Measuring the Performance of Parallel Message-based Process Architectures," in *Proceedings of the Conference on Computer Communications (INFOCOM)*, (Boston, MA), pp. 624–633, IEEE, April 1995.

[7] 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.

[8] D. C. Schmidt and C. Cleeland, "Applying a Pattern Language to Develop Extensible ORB Middleware," in *Design Patterns in Communications* (L. Rising, ed.), Cambridge University Press, 2000.

[9] D. C. Schmidt, "Applying Design Patterns and Frameworks to Develop Object-Oriented Communication Software," in *Handbook of Programming Languages* (P. Salus, ed.), MacMillan Computer Publishing, 1997.

[10] D. C. Schmidt and P. Stephenson, "Experiences Using Design Patterns to Evolve System Software Across Diverse OS Platforms," in *Proceedings of the* 9^{th} *European Conference on Object-Oriented Programming*, (Aarhus, Denmark), ACM, August 1995.

[11] M. A. Linton, J. Vlissides, and P. Calder, "Composing User Interfaces with InterViews," *IEEE Computer*, vol. 22, pp. 8–22, February 1989.

[12] D. Ritchie, "A Stream Input–Output System," *AT&T Bell Labs Technical Journal*, vol. 63, pp. 311–324, Oct. 1984.

[13] W. R. Stevens, *UNIX Network Programming, First Edition*. Englewood Cliffs, NJ: Prentice Hall, 1990.

[14] S. Rago, *UNIX System V Network Programming*. Reading, MA: Addison-Wesley, 1993.

[15] R. H. Halstead, Jr., "Multilisp: A Language for Concurrent Symbolic Computation," *ACM Trans. Programming Languages and Systems*, vol. 7, pp. 501–538, Oct. 1985.

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