Using Design Patterns to Develop High-Performance Object-Oriented Communication Software Frameworks

Douglas C. Schmidtla
schmidt@cs.wustl.edu
Department of Computer Science
Washington University, St. Louis 63130
(TEL) 314-935-7538, (FAX) 314-935-7302

This paper appeared in the Proceedings of the 8^{th} Annual Software Technology Conference, Salt Lake City, Utah, April 21–26, 1996.

1 Introduction

Developing extensible communication software that effectively utilizes concurrency on parallel platforms is a complex task. Despite dramatic increases in network and host performance, the design and implementation of communication software remains a challenging problem. Moreover, the growing heterogeneity of hardware/software architectures and diversity of operating system platforms often make it hard to directly reuse existing algorithms, detailed designs, interfaces, or implementations [1].

Two promising techniques for alleviating communication software complexity are *design patterns* and *object-oriented frameworks*. Design patterns capture the static and dynamic structures and collaborations among components in successful solutions to problems that arise when building software [2]. They help to enhance software quality by addressing fundamental challenges in large-scale system 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 experience.

An object-oriented communication framework is an integrated collection of components that cooperate to define a reusable architecture for a family of related communication systems [3]. A framework provide a set of "semi-complete" applications that automate common communication software tasks (such as event demultiplexing, event handler dispatching, connection establishment, routing, configuration of application services, and concurrency control [4]).

The emerging focus on design patterns and communication frameworks in the object-oriented community offers software developers both a language of discourse and a set of directly reusable software components for capturing the essence of successful architectures, components, policies, services, and programming mechanisms. Once expressed in the pattern form, reusable communication frameworks can be recast in new contexts to facilitate the widespread reuse of software architectures, detailed designs, algorithms, and implementations.

The remainder of this paper is organized as follows: Section 2 outlines how patterns and frameworks have been applied to create a reusable object-oriented software architecture for high-performance application-level `Gateways`; Section 3 outlines the components in a reusable communication software framework used to build application-level `Gateways`; Section 4 examines the design patterns that form the basis for the framework and `Gateways`; Section 5 compares these patterns with those described in related work; and Section 6 presents concluding remarks.

2 A Reusable Object-Oriented Software Architecture for Application-level Gateways

This paper presents a case study illustrating how design patterns and frameworks are being applied in practice to facilitate widespread reuse of software experience in production communication systems. Patterns aid the development of reusable components and frameworks in these systems by capturing the structure and collaboration of participants in a software architecture at a higher level than (1) source code and (2) object-oriented design models that focus on individual objects and classes. Thus, patterns enable widespread reuse of software architecture, even when reuse of algorithms, implementations, interfaces, or detailed designs is not feasible [1]. Likewise, frameworks can be viewed as concrete realizations of design patterns that facilitate direct reuse of design and code. The particular system described in this case study is a reusable object-oriented software architecture for high-performance application-level `Gateways`.

2.1 System Overview

An application-level `Gateway` routes messages between `Peers` in a communication system (shown in Figure 1). The `Gateway` serves as a Mediator [5] that decouples co-

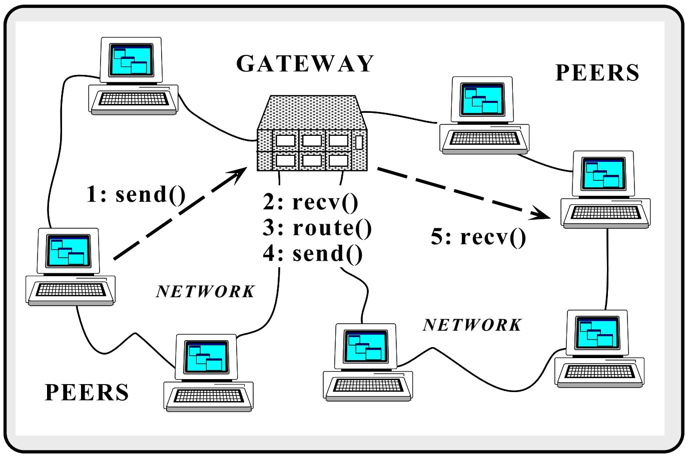

Figure 1: The Structure and Collaboration of Peers and the Gateway

operating components in a software system and allows them to interact without having direct dependencies on each other [6]. Messages routed through a `Gateway` typically contain payloads such as commands, status messages, and bulk data exchanged by `Peers`. These payloads are encapsulated in routing messages.

This paper presents the object-oriented architecture and design of application-level `Gateways` in terms of the *strategic* design patterns and framework components used to guide their construction. Strategic design patterns have an extensive impact on the software architecture for solutions in a particular domain. For example, the Router pattern described in Section 4.4 decouples input mechanisms from output mechanisms to ensure that message processing is not disrupted or postponed indefinitely when a `Gateway` experiences congestion or failure. This pattern greatly simplifies the design and quality of service in single-threaded `Gateways` that use connection-oriented protocols such as TCP/IP or IPX/SPX. Application-level `Gateway` also utilize many tactical patterns (such as Factories and Iterators [5]). Tactical patterns have a relatively localized impact on a software architecture compared with strategic patterns (which have more sweeping implications on software architecture).

Due to stringent requirements for reliability, performance, and extensibility, application-level `Gateways` serve as excellent exemplars for presenting the structure, participants, and consequences of design patterns that appear in many communication software systems. Figure 2 illustrates the structure, associations, and internal and external collaborations among objects within a reusable software architecture for application-level `Gateways`.[1] This architecture is based on extensive experience developing connection-oriented `Gateways` for various commercial and academic projects.

2.2 Impact of Patterns and Frameworks on Software Reuse

After building a range of communication systems, it became clear that the software architecture of application-level `Gateways` was largely independent of the protocols used to route messages to `Peers`. This realization enabled the components depicted in Figure 2 to be reused on many communication software projects. The ability to reuse these components so widely stems from two factors:

1. *Understanding the strategic design patterns within the domain of communication software* – Some of these patterns have been documented individually in the patterns literature (such as the Reactor [8], Connector [9], and the Acceptor [10]). Section 4 describes several of these patterns in terms of an integrated system of design patterns that characterize the structure and collaboration of communication software patterns in the context of application-level `Gateways`.
2. *Implementing an object-oriented framework that implements these common patterns* – The systems described in this paper were implemented with the ADAPTIVE Communication Environment (ACE) software [4]. The ACE framework implements a collection of design patterns that recur when building concurrent and reactive [8] communication software. ACE provides a rich set of reusable C++ wrappers, class categories, and frameworks that perform common communication software tasks (such as event demultiplexing, event handler dispatching, connection establishment, routing, dynamic configuration of application services, and concurrency control).

The design patterns and framework components described in this paper have been used extensively throughout large-scale telecommunication and electronic medical imaging projects [1, 11], as well as on academic research projects [4]. Both the patterns and the ACE components evolved continuously over time via a continual process of "round trip gestalt" [7].

3 An Object-Oriented Framework for the Gateway

This section describes how various communication components in the ACE framework were reused and extended to implement the application-level `Gateway` architecture. Following this overview, Section 4 examines the family of design patterns that underly these reusable components.

[1] Relationships between components are illustrated throughout this paper using Booch notation [7]. In these figures solid clouds indicate objects; nesting indicates composition relationships between objects; and undirected edges indicate some type of link exists between two objects. Dashed clouds indicate classes; directed edges indicate inheritance relationships between classes; and an undirected edge with a small circle at one end indicates either a composition or uses relation between two classes; and a dashed directed edge indicates template instantiation.

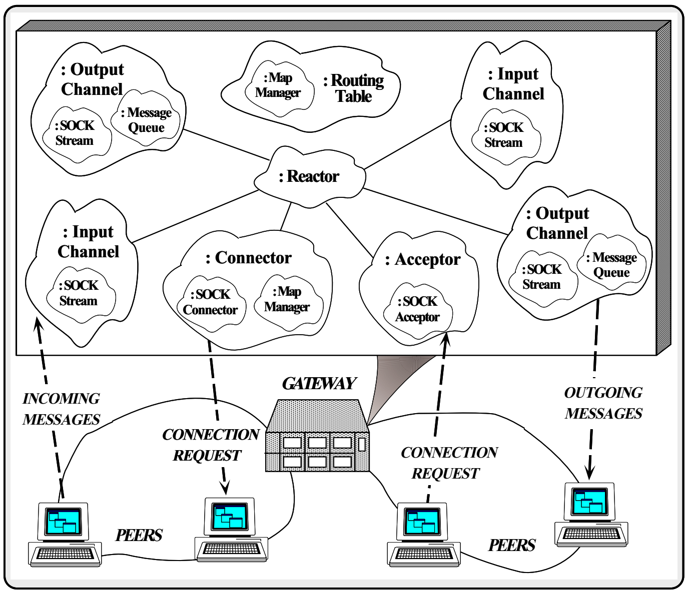

Figure 2: The Object-Oriented `Gateway` Software Architecture

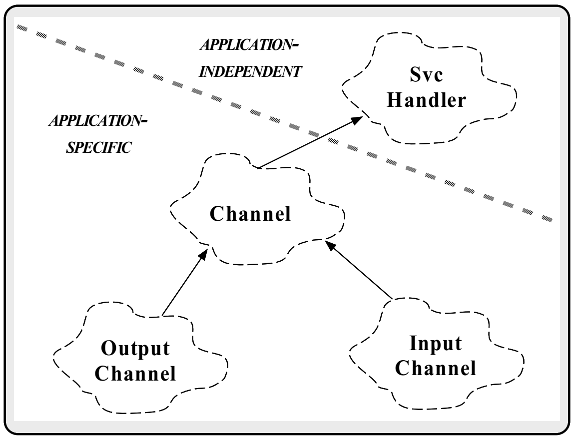

Figure 3: Channel Inheritance Hierarchy

3.1 Applying ACE Components to the Gateway

The primary ACE components used in the `Gateway` include the `Reactor` [8], which encapsulates the UNIX `select` event demultiplexing system call; `SOCK Stream`, `SOCK Connector`, and `SOCK Acceptor` [11], which encapsulate the socket network programming interface; and `Map Manager` and `Message Queue` [4], which manage communication messages efficiently. These components and their use in the `Gateway` are described below.

• **Reactor:** The `Reactor` [8] is a reusable object-oriented event demultiplexing mechanism based on the Reactor pattern (outlined in Section 4.1). It channels all external event stimuli in a `Gateway` to a single demultiplexing point. This permits single-threaded `Gateways` to wait on events handles, demultiplex events, and dispatch event handlers efficiently. An event indicates to a `Gateway` that something significant has occurred (*e.g.,* the arrival of a new connection or work request). The primary source of `Gateway` events is routing messages that encapsulate various payloads (such as commands, status messages, and bulk data). The `Reactor` provides a coarse-grained form of concurrency control for a single-threaded `Gateway`. It 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 also minimizes context switching.

• **Input Channel and Output Channel:** These classes implement the Router pattern (described in Section 4.4). Both classes inherit from a common ancestor: base class `Channel` (shown in Figure 3). This enables them to communicate with `Peers` via an ACE `SOCK Stream` object provided by the `Channel` base class. `Input Channels` are responsible for routing incoming messages to their destination(s). The `Reactor` notifies an `Input Channel` when it detects an event on that channel's `SOCK Stream` endpoint. The `Input Channel` then receives and frames a routing message from that endpoint, consults the `Routing Table` to determine the set of `Output Channel` destinations for the message, and asks the selected `Output Channels` to forward the message to the appropriate `Peer` destination(s).

An `Output Channel` is responsible for reliably delivering routing messages to their destinations. 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 ensures that 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 `Output Channel` to stop producing messages until the destination `Peer` consumes the data.

A reusable ACE `Message Queue` object chains together unsent messages in the order they must be delivered when flow control mechanisms permit. Once the flow control window opens up, the `Reactor` calls back to the `handle_event` method of the `Output Channel`. This signals the channel to start 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.

• **Routing Table:** `Input Channels` use the `Routing Table` to map addressing information contained in routing messages sent by `Peers` to the appropriate set of `Output Channels`. The `Routing Table` reuses the ACE `Map Manager` collection class. A `Map Manager` is a parameterized collection that efficiently maps external ids (*e.g.,* `Peer` routing addresses) onto internal ids (*e.g.,* `Output Channels`).

• **Channel Connector and Channel Acceptor:** The `Channel Connector` and `Channel Acceptor` are reusable Factories [5] used by the `Gateway` to actively and passively establish connections with `Peers` and produce the connected `Input Channels` and `Output Channels` described above. These components are based on the Connector pattern (described in Section 4.2) and Acceptor pattern (described in Section 4.3).

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

1. From the `Gateway` to the `Peers` – which is typically done when the `Gateway` first starts up to establish the initial system configuration of `Peers`.
2. From a `Peer` to the `Gateway`– which is typically done once the system is running when a new `Peer` wants to send or receive routing messages.

In a large system several hundred `Peers` may be connected to a single `Gateway`. To expedite connection setup initiated from the `Gateway` to all these `Peers`, the `Gateway` uses the asynchronous connection mechanisms provided by the `Channel Connector` and its underlying ACE `SOCK Connector` [11]. When a `SOCK Connector` connects two socket endpoints via TCP it produces a `SOCK Stream` object, which is used to exchange data between that `Peer` and the `Gateway`.

To decrease connection establishment latency, the `Gateway`'s `Channel Connector` initiates all connections asynchronously rather than connecting each `Peer` synchronously. Asynchrony helps decrease connection latency over long delay paths (such as wide-area networks (WANs) build over satellites or long haul terrestrial links).

3.2 Motivation for Using ACE

To enhance performance and interoperability, as well as to reuse existing tracking station software and hardware, connections between the `Gateway` control facility applications and tracking stations are implemented using the TCP/IP communication protocol suite. In particular, the `Gateway` does *not* higher-level distributed object computing tools like CORBA [12] for its communication infrastructure. There are several reasons for this decision:

- The performance of CORBA implementations has generally not been optimized to eliminate key sources of communication overhead for transmitting bulk data over high-speed, long-delay networks [11, 13]. This overhead stems from non-optimized presentation layer conversions, data copying, and memory management, inefficient receiver-side demultiplexing and dispatching operations, synchronous stop-and-wait flow control, and non-adaptive retransmission timer schemes.
- CORBA is not well suited to handle the peer-to-peer, asynchronous behavior of the `Gateway`. In particular, many CORBA implementations do not support non-blocking method invocations (even for `oneway` operations). The problem is that flow control mechanisms provided by the *de facto* CORBA transport protocol (TCP) may indefinitely block a method that outputs messages. TCP flow control ensures that a fast producer does not send data faster than a slower consumer can buffer and process the data. If the consumer runs out of buffer space TCP instructs the producer to stop transmitting until the consumer removes the data from the OS buffer layer. Many versions of CORBA block a sender when a TCP connection encounters flow control. Therefore, it is hard to write a robust, single-threaded application that will not hang indefinitely.
- Legacy communication applications and protocol stacks do not conform to the CORBA interface nor its wire protocol [14]. When combined with the output blocking problem described above, the level of effort required to port legacy applications to CORBA clearly exceeds the benefits of using a single OO communication infrastructure.

Since the use of CORBA was infeasible, the ACE framework was combined with the design patterns described below to build a robust, extensible, and high-performance `Gateway`.

4 A System of Design Patterns for the Gateway

A *design pattern* is a recurring solution to a design problem within a particular domain (such as business data processing, telecommunications, graphical user interfaces, databases, or distributed communication software). A design pattern description typically conveys the following information [5]:

- The intent of the pattern
- The design forces that motivate the pattern
- The solution to these forces
- The related classes and their roles in the solution
- The responsibilities and dynamic collaborations among classes
- The positive and negative consequences of using the pattern
- Guidance for implementors of the pattern
- Example source code illustrating how the pattern is applied
- References to related work.

A *family of design patterns* (also called a "pattern language" [15] or a "pattern system" [6]) is a set of related patterns that collaborate to solve a broader set of problems that arise in a domain. A pattern family *description* illustrates how the constituent patterns interact to form a web of design solutions [6]. Figure 4 illustrates the key strategic and

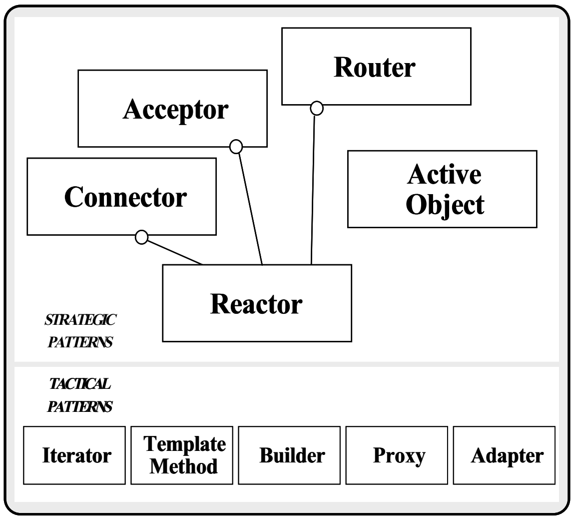

Figure 4: The Family of Patterns for the `Gateway`

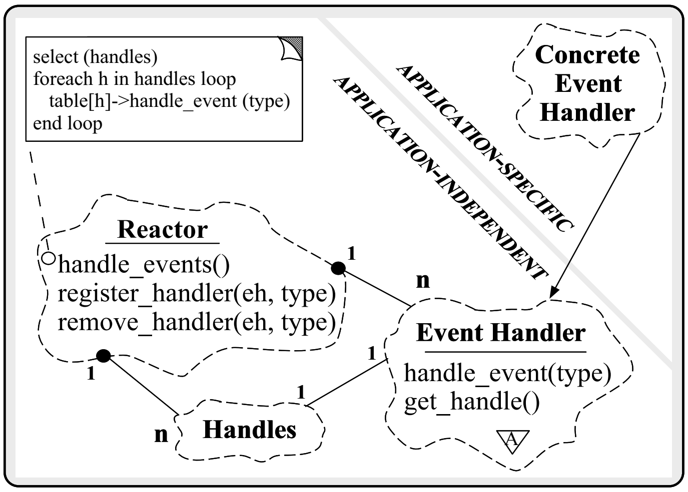

Figure 5: Structure and Participants in the Reactor Pattern

tactical patterns in a family of patterns for singled-threaded, connection-oriented application-level `Gateways`. The following four strategic patterns related to connection-oriented, application-level `Gateways` are examined in this section:

- *The Reactor pattern* – decouples event demultiplexing and event handler dispatching from services performed in response to events;
- *The Connector pattern* – decouples active service initialization from the tasks service performed once the service is initialized.
- *The Acceptor pattern* – decouples passive service initialization from the tasks performed once the service is initialized;
- *The Router pattern* – decouples input mechanisms from output mechanisms to prevent blocking in a single-threaded `Gateway`.

These patterns form the family of patterns underlying the object-oriented software architecture of application-level `Gateways` described in Section 2. This paper focuses on these strategic patterns since they are crucial to the architecture, design, and implementation of communication `Gateways`. Moreover, these strategic patterns express design expertise that can be reused across a broad range of communication software. This family of patterns was discovered based on extensive design and implementation experience with communication systems (including on-line transaction processing systems [16], telecommunication switch management systems [1], electronic medical imaging systems [11], and parallel communication subsystems [4]).

Due to space limitations, the strategic `Gateway` patterns are not described as thoroughly as the patterns in catalogs such as [5, 6], nor are sample implementations provided. Likewise, the *tactical* patterns shown in Figure 4 are not described in detail either. In contrast to strategic patterns (which are often domain-specific and have sweeping design implications), tactical patterns are generally domain-independent and have a relatively localized impact on a software architecture. For instance, Iterator [5] is a tactical pattern used in the `Gateway` to allow `Channels` in the `Routing Table` to be processed 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 architecture as strongly as the strategic patterns described in this paper. Other tactical patterns used extensively throughout the `Gateway` include the following:

- *Factory Methods* – which decouple object creation from object use.
- *Iterators* – which decouple sequential access to a container from the representation of the container.
- *Adapters* – which encapsulate existing procedural interfaces to make them object-oriented.
- *Template Method* – where an algorithm is written such that some steps are supplied by a derived class.

As described below, many of these tactic patterns form the basis for the strategic patterns presented in this paper.

4.1 The Reactor Pattern

Intent: The Reactor pattern decouples event demultiplexing and event handler dispatching from the services performed in response to events.

Forces: The Reactor pattern resolves the following forces that impact the design of event-driven communication software:

1. *The need to demultiplex multiple types of events from multiple sources of events efficiently within a single*

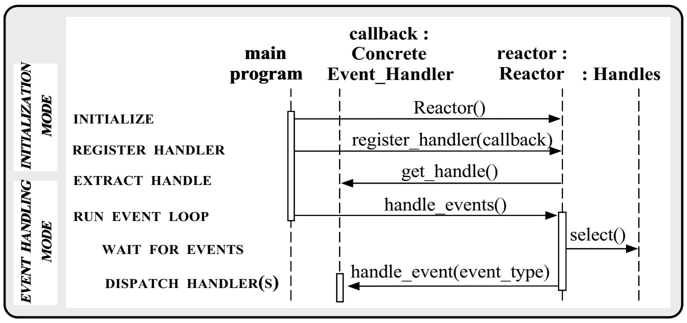

Figure 6: Object Interaction Diagram for the Reactor Pattern

thread of control – A Reactor serializes application event handling within a process at the level of event demultiplexing. By using the Reactor pattern the need for more complicated threading, synchronization, or locking within an application is often eliminated.

2. *The need to extend application behavior without requiring changes to the event dispatching framework* – The Reactor factors out the demultiplexing and dispatching mechanisms (which are independent of an application and thus reusable) from the event handler processing policies (which are specific to an application).

Structure and Participants: 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. An 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 events.

An `Event Handler` specifies an abstract interface used by the `Reactor` to dispatch callback methods defined by objects that register to handle input, output, signal, and timeout events of interest. Each `Concrete Event Handler` selectively implements callback method(s) to process events in an application-specific manner.

Collaborations: Figure 6 illustrates the collaborations between participants in the Reactor pattern. These collaborations are divided into the following two phases:

1. *Initialization phase* – where `Concrete Event Handler` objects are registered with the `Reactor`
2. *Event handling phase* – where methods on the objects are called back to handle particular types of events.

Uses: Figure 7 outlines how the `Reactor` is used in a `Gateway`. A `Reactor` object dispatches incoming routing messages to the associated `Input Channel`, where they are routed to `Output Channels`. The Reactor also ensures that outgoing routing messages are eventually delivered on flow controlled `Output Channels` (described in Section 4.4). In addition, the `Reactor` dispatches events that

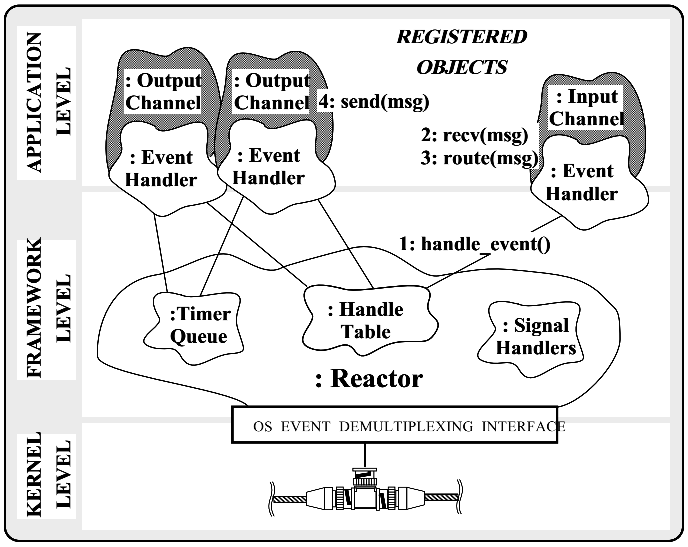

Figure 7: Using the Reactor Pattern in the `Gateway`

indicate the completion status of connections actively initiated asynchronously (used by the `Channel Connector` described in Section 4.2), as well as to accept passively initiated connections (used by the `Channel Acceptor` described in Section 4.3).

The Reactor pattern has been used in many single-threaded event-driven frameworks (such as the Motif, Interviews [17], System V STREAMS [18], the `ASX` object-oriented communication framework [4], and implementations of DCE and CORBA). In addition, it forms as the foundation for the other strategic patterns for application-level `Gateways` presented below.

4.2 The Connector Pattern

Intent: The Connector pattern decouples active[2] service initialization from the tasks performed once a service is initialized.

Forces: The Connector pattern resolves the following forces that impact the design of connection-oriented communication software (particularly clients) when using lower-level network programming interfaces (like sockets [19] and TLI [20]):

1. *The need to reuse active connection establishment code for each new service* – The Connector pattern permits key characteristics of services (such as the concurrency strategy or the data format) to evolve independently and transparently from the mechanisms used to establish the connections. Since service characteristics change more frequently than connection establishment mechanisms

[2] Communication software is typified by asymmetric roles for establishing connections between clients and servers. In general, servers (who play a *passive* role) listen for clients (who play an *active* role) to initiate connections.

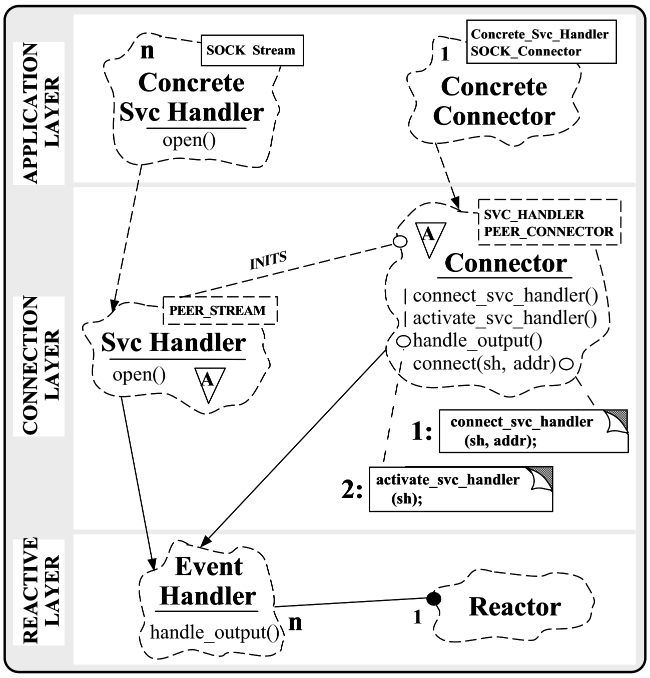

Figure 8: Structure and Participants in the Connector Pattern

this separation of concerns helps reduce software coupling and increases code reuse.

2. *The need to make the connection establishment code portable across platforms that contain different network programming interfaces* – This is particularly important for asynchronous connection establishment, which is hard to program portably and correctly using lower-level network programming interfaces (such as sockets and TLI).

3. *The need to actively establish connections with large number of peers efficiently* – The Connector pattern can employ asynchrony to initiate and complete multiple connections in non-blocking mode. By using asynchrony, the Connector pattern enables applications to actively establish connections with a large number of peers efficiently over long-delay WANs.

4. *The need to enable flexible service concurrency policies* – Once a connection is established, peer applications use the connection to exchange data to perform some type of service (*e.g.,* remote login, WWW HTML document transfer, etc.). A service can run in a single-thread, in multiple threads, or multiple processes, regardless of how the connection was established.

Structure and Participants: Figure 8 illustrates the structure and participants in the Connector pattern. As shown in the figure, the participants in this pattern leverage off the Reactor pattern by inheriting from its `Event Handler` interface. Using the Reactor pattern enables multiple connections to be actively established asynchronously within a single thread of control.

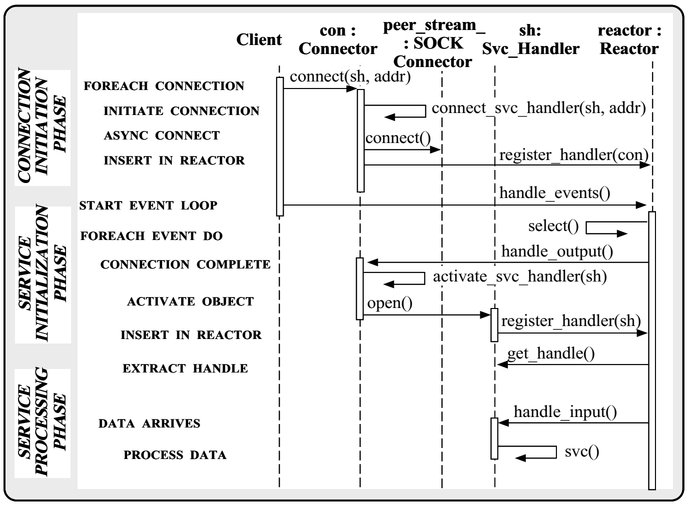

Figure 9: Object Interaction Diagram for the Connector Pattern

The `Connector` is a factory that assembles the resources necessary to create, connect, and activate a service handler. In addition, it implements the strategy for establishing connections with `Peers` asynchronously. It is parameterized by a particular type of `PEER CONNECTOR` and `SVC HANDLER`. The `PEER CONNECTOR` supplies the underlying transport mechanism (such as C++ wrappers for sockets or TLI) used to actively establish the connection asynchronously. The `SVC HANDLER` specifies an abstract interface for defining a service that communicates with a connected `Peer`. Moreover, a `SVC HANDLER` is parameterized by a `PEER STREAM` endpoint. The `Connector` connects this endpoint to its `Peer` when a connection is established successfully. Note that by inheriting from `Event Handler`, a `SVC HANDLER` can register with a `Reactor` in order to demultiplex data within a single event-driven thread of control.

Parameterized types are used to decouple the Connector pattern's connection establishment strategy from the type of service and the type of connection mechanism. Developers produce `Concrete Connectors` by supplying arguments for these types. This enables the wholesale replacement of these types, without affecting the Connector pattern's connection establishment strategy.

Collaborations: The collaborations among participants in the Connector pattern are divided into three phases:

1. *Connection initiation phase* – which actively connects one or more `Svc Handlers` with their peers. Connections can either be initiated synchronously or asynchronously. The `Connector` determines the *strategy* for actively establishing connections.

2. *Service initialization phase* – which activates a `Svc Handler` by calling its `open` method when the connection associated with it completes successfully. The

open method of the Svc Handler 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. Depending on the open method of Svc Handler, this phase may employ the Reactor pattern (or some other type of concurrency mechanisms such as Active Objects [21]) to process incoming events. For example, when commands arrive at a Command Handler in the Gateway, the Reactor dispatches Event Handlers to frame the commands, determine outgoing routes, and deliver the commands to their destinations.

Figure 9 illustrates the collaboration among participants in the Connector pattern using *asynchronous* connection establishment.

Uses: The Gateway uses the Connector pattern to simplify the task of connecting to a large number of Peers. During Gateway initialization, a list of Peer port addresses are read from a configuration file. These addresses are bound to dynamically created Channels (which inherit from Svc Handler). All connections are then initiated asynchronously and the connections are completed in parallel.

Figure 10 illustrates the relationship between participants in the Connector pattern after four connections have been established. Three other connections that have not yet completed are owned by the Connector. As shown in this figure, the Connector maintains a table of the three Channels whose connections are pending completion. As connections complete, the Connector removes the connected Channel from its table and activates it. Once activated, Input Channels register themselves with the Reactor. Henceforth, when routing messages arrive, Input Channels receive and forward them to Output Channels, which deliver the messages to their destinations (these activities are described in Section 4.4). Input Channels and Output Channels are objects residing in the Gateway. In contrast, the original source and the intended destination(s) of routing messages reside on other hosts across the network.

In addition to establishing connections, a Gateway can use the Connector in conjunction with the Reactor to ensure that connections are restarted when 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 an Output Channel due to network congestion). If a connection fails unexpectedly, an exponential-backoff algorithm can be implemented using the Reactor to restart the connection efficiently.

4.3 The Acceptor Pattern

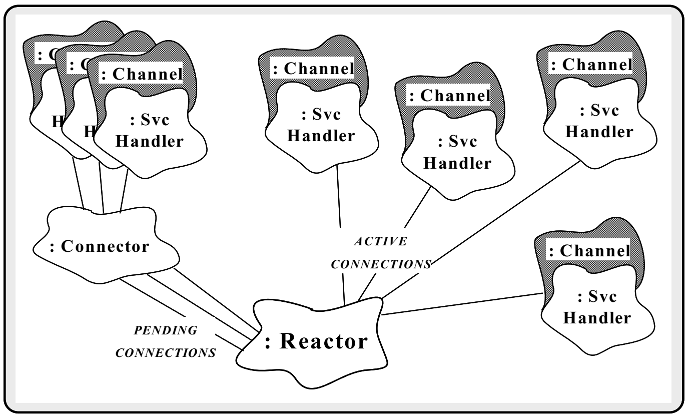

Figure 10: Using the Connector Pattern in the Gateway

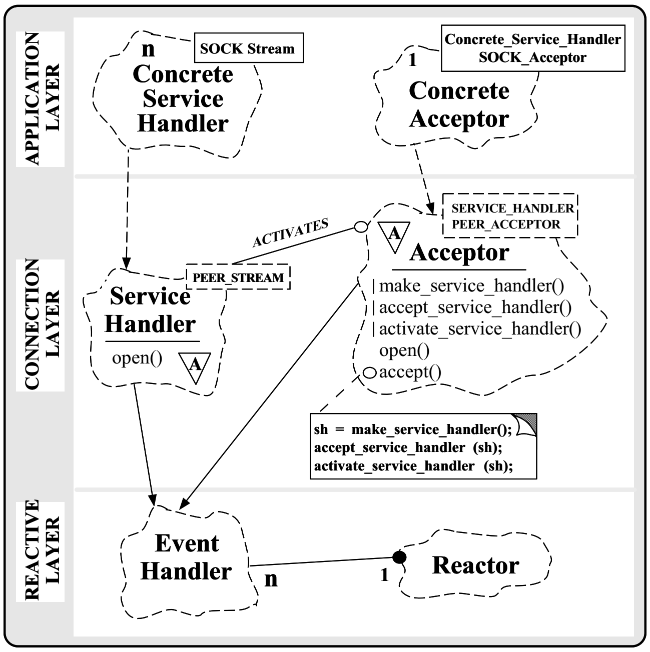

Figure 11: Structure and Participants in the Acceptor Pattern

Intent: The Acceptor pattern decouples passive service initialization from the tasks performed once the service is initialized.

Forces: The Acceptor pattern resolves the following forces that impact the design of connection-oriented communication software (particularly servers) when using lower-level network programming interfaces (like sockets [19] and TLI [20]):

1. *The need to reuse passive connection establishment code for each new service* – The Acceptor pattern permits key characteristics of services (such as the concurrency strategy or the data format) to evolve independently and transparently from the mechanisms used to establish the connections. Since service characteristics change more frequently than connection establishment mechanisms this separation of concerns helps reduce software coupling and increases code reuse.
2. *The need to make the connection establishment code portable across platforms that contain different network programming interfaces* – Parameterizing the Acceptor'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* – Once a connection is established, peer applications use the connection to exchange data to perform some type of service (*e.g.,* remote login, WWW HTML document transfer, etc.). A service can run in a single-thread, in multiple threads, or multiple processes, regardless of how the connection was established.
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 `Connector` from the `Svc Handler` passive-mode listener endpoints cannot accidentally be used incorrectly (*e.g.,* to try to read or write data on a passive-mode listener socket used to accept connections).

The Acceptor pattern is the "dual" of the Connector pattern described in Section 4.2. Unlike the Connector pattern (which establishes connections *actively*), the Acceptor pattern establishes connections *passively*.

Structure and Participants: Figure 11 illustrates the structure and participants in the Acceptor pattern. This pattern leverages off the Reactor pattern's `Reactor` to passively establish multiple connections within a single thread of control. The `Acceptor` implements the strategy for establishing connections with `Peers`. It is parameterized by concrete types that conform to the interfaces of the formal template arguments `SVC HANDLER` (which performs a

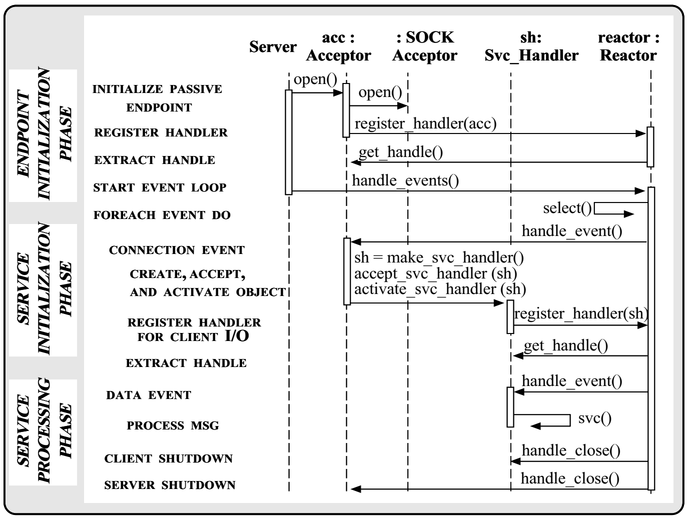

Figure 12: Object Interaction Diagram for the Acceptor Pattern

service in conjunction with a connected `Peer`) and `PEER ACCEPTOR` (which is the underlying mechanism used to passively establish the connection). The `Svc Handler` shown in Figure 11 is a concrete type that defines the interface for an application-specific service. It inherits from `Event Handler` (shown in Figure 5), which allows it to be dispatched by the `Reactor` when connection events occur. In addition, `Svc Handler` is parameterized by a `PEER STREAM` endpoint. The `Acceptor` associates this endpoint with its `Peer` when a connection is established successfully.

As with the Connector pattern, parameterized types are used to enhance portability since the Acceptor pattern's connection establishment strategy is independent of the type of service and the type of IPC mechanism. Programmers supply concrete arguments for these types to produce a `Concrete Acceptor`. 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 [5]. Parameterized types were used to implement this pattern since they improve run-time efficiency at the expense of additional compile-time and link-time time and space overhead.

Collaboration: Figure 12 illustrates the collaboration among participants in the Acceptor pattern. These collaborations are divided into 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 then goes into an event loop waiting on that endpoint for connection requests to arrive from peers.
2. *Service activation phase* – Since an `Acceptor` inherits from an `Event Handler` the `Reactor` can

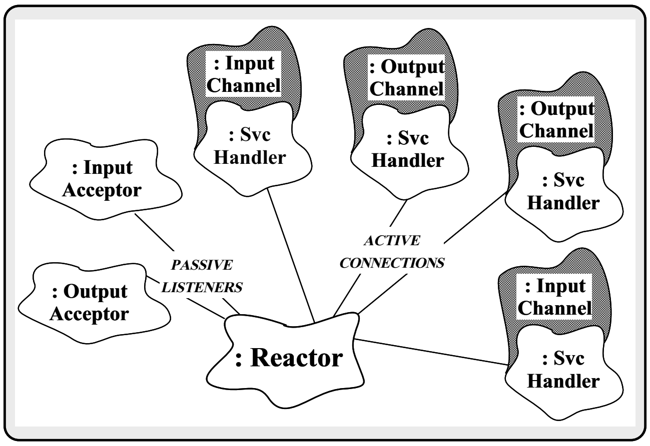

Figure 13: Using the Acceptor Pattern in the `Gateway`

dispatch the `Acceptor`'s `handle_event` method when connection events arrive. When connections arrive, the `Reactor` calls back to the `Acceptor`'s `handle_event` method. This Template Method [5] performs the `Acceptor`'s `Svc Handler` activation strategy. This strategy assembles the resources necessary to create a new `Concrete Svc Handler` object, accept the connection into this object, and activate the `Svc Handler` by calling its `open` method.

3. *Service processing phase* – once activated, the `Svc Handler` processes incoming event messages arriving on the `PEER STREAM`. A `Svc Handler` will process incoming event messages using the Reactor pattern or some other form of concurrent event handling such as the Active Object pattern [21]. The concurrency strategy used by a `Svc Handler` is defined by its `open` method.

Uses: Figure 13 illustrates how the Acceptor pattern is used by the `Gateway`. The `Gateway` uses this pattern when it plays the passive connection role. In this case, the `Peers` connect to `Gateway`, which uses the Acceptor pattern to decouple the activity of connecting passively from the routing service provided once the connection is established.

The intent and general architecture of the Acceptor pattern is also found in network server management tools like `inetd` [19] and `listen` [20]. 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.

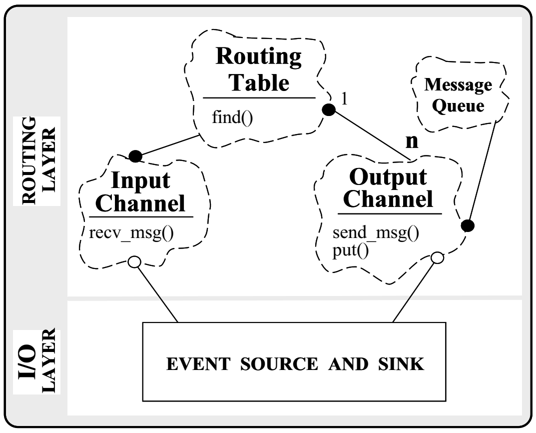

Figure 14: Structure and Participants in the Router Pattern

4.4 The Router Pattern

Intent: The Router pattern decouples multiple sources of input from multiple sources of output to prevent blocking in a single-threaded `Gateway`.

Forces: This pattern resolves the following force that impacts the design of single-threaded connection-oriented `Gateways`.

1. *The need to prevent misbehaving connections from disrupting the quality of service for well-behaved connections* – It is paramount that `Gateway` message routing is not disrupted or postponed indefinitely when congestion or failure occurs on incoming and outgoing links. For example, if outgoing connections flow control due to network congestion or `Peer` failure, the `Gateway` must not perform blocking `send` operations on any single channel. Otherwise, messages on other channels could not be sent or received and the end-to-end quality of service provided to `Peers` would degrade.
2. *The need to allow different concurrency strategies for Input and Output Channels* – although this paper focuses on single-threaded `Gateways` there are alternative concurrency strategies such as (1) spawning a separate thread for every `Input Channel` and `Output Channel`, (2) spawning a thread for each `Output Channel` but multiplexing all `Input Channels` in a single thread, or (3) using a pool of pre-spawned threads. Different strategies are appropriate under different situations, depending on factors such as the number of CPUs, context switching overhead, and number of `Peers`. By decoupling `Input Channels` from `Output Channels` the Router pattern allows various concurrency strategies to be configured flexibly into a `Gateway`.

Structure and Participants: Figure 14 illustrates the structure and participants in the Router pattern. As with the Connector pattern, the Router pattern uses a `Reactor` to allow multiple events on different connections to be demultiplexed within a single thread of control. The Input Channels and Output Channels inherit indirectly from Event Handler. This enables the Reactor to dispatch their handle_event methods when messages arrive and flow control conditions subside, respectively. An Input Channel uses a Routing Table to map routing messages onto one or more Output Channels. Since the Input Channels are separate from the Output Channels their implementations may vary independently. This decoupling is important since it allows different concurrency strategies to be used for input and output.

Although TCP connections are 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. First, it simplifies the construction of Gateway Routing Tables. Second, it allows more flexibility in connection configuration and concurrency strategies. Finally, it enhances reliability if errors occur on a connection (since Input and Output Channels can be reconnected independently).

Collaborations: Figure 15 illustrates the collaboration among participants in the Router pattern. These collaborations may be divided into three phases:

1. *Input processing* – in this phase Input Channels use non-blocking I/O to incrementally reassemble incoming TCP segments into complete routing messages;
2. *Route selection* – in this phase Input Channels consult a Routing Table to select the Output Channels responsible for sending the routing messages;
3. *Output processing* – in this phase the selected Output Channels transmit the routing messages to their destination(s) without blocking the process.

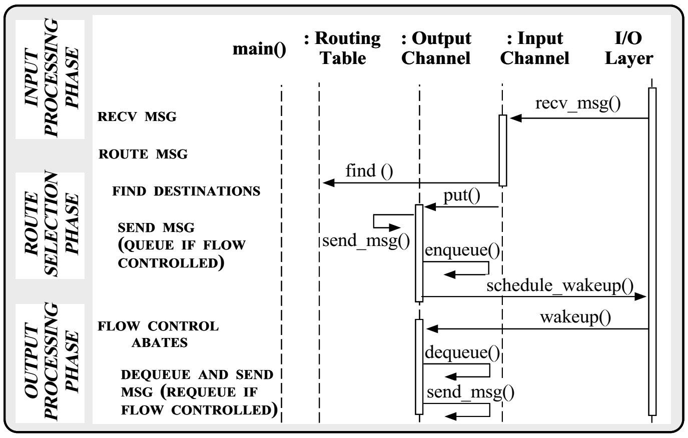

Figure 15: Object Interaction Diagram for the Router Pattern

Uses: Figure 16 illustrates how the Router pattern is used in the Gateway. Input Channel and Output Channel processing routes messages within a single thread of control by using the Reactor object. The use of single-threading eliminates the overhead of synchronization (since access to shared objects like the Routing Table need not be serialized) and context switching (since message routing occurs in a single thread).

The primary challenge of building a reliable, single-threaded, connection-oriented Gateway revolves around avoiding blocking I/O. This is necessary to reliably manage flow control on Output Channels. If the Gateway were to block indefinitely when sending on a congested connections incoming messages could not be routed, even if those messages were destined for non-flow controlled Output Channels.

Figure 16 illustrates the sequence of collaborations between Router pattern participants in a single-threaded Gateway. First, Input and Output Channel descriptors are set into non-blocking mode after the Connector activates them. Message are subsequently received in fragments by Input Channels. When an Input Channel successfully receives and frames an entire message it uses the Routing Table to determine the appropriate set of Output Channels. It then passes the message to these Output Channels, which try to send the message to the destination Peer.

To avoid blocking, all send operations in Output Channels must check to see flow control is enabled. If not, an entire message can be sent successfully (depicted by the Output Channel in the upper right-hand corner of Figure 16). The Router pattern must use a different strategy, however, when a send encounters a flow controlled connection (depicted by the Output Channel in the lower right-hand corner of Figure 16).

To handle flow control, the Output Channel inserts the message it is trying to send into its Message Queue. It then instructs the Reactor to call back to the Output Channel 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 Output Channel, 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 Router pattern – it never blocks on any single I/O channel.

5 Related Work

[5, 6, 22] identify, name, and catalog many fundamental object-oriented design patterns. This section examines how the patterns described in this paper relate to other patterns in

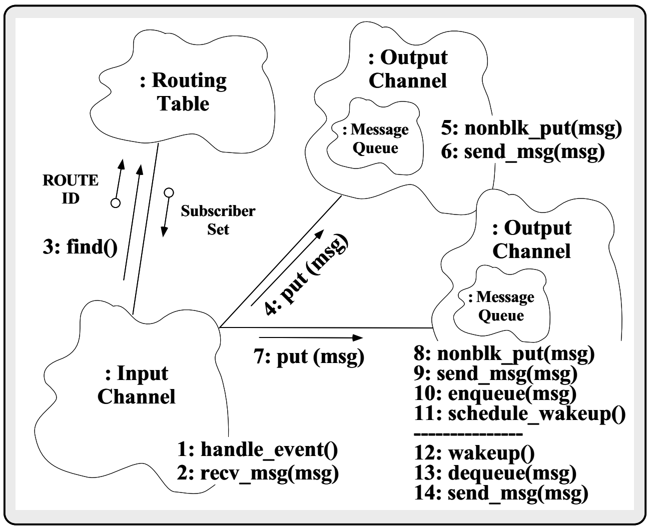

Figure 16: Using the Router Pattern in the `Gateway`

the literature.

The Reactor pattern is related to the Observer pattern [5]. In the Observer pattern, *multiple* dependents are updated automatically when a subject changes. In the Reactor pattern, a single handler is dispatched automatically when an event occurs. Thus, For each event the Reactor dispatches a *single* handler (though there can be multiple sources of events). The Reactor pattern also provides a Facade [5]. The Facade pattern presents an interface that shields applications from complex object relationships within a subsystem. The Reactor pattern shields applications from complex mechanisms that perform event demultiplexing and event handler dispatching.

The mechanism the Reactor uses to dispatch `Event Handlers` is similar to the Factory Callback pattern [23]. The intent of both patterns is to decoupling event reception from event processing. The primary different is that the Factory Callback is a creational pattern, whereas the Reactor dispatching is a behavioral pattern.

The Connector pattern is a variation of the Template Method and Factory Method patterns [5]. 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 Connector pattern is a Factory that use Template Methods to create, connect, and activate handlers for communication channels. In the Connector pattern, the `connect` method implements a standard algorithm for initiating a connection and activating a handler when the connection is established. The intent of the Connector pattern is similar to the Client/Dispatcher/Server pattern described in [6]. They both are concerned with separating active connection establishment from the subsequent service. The primary difference is that the Connector pattern addresses both synchronous and asynchronous connection establishment.

The Acceptor pattern can also be viewed as a variation of the Template Method and Factory Method patterns [5]. The Acceptor pattern is a connection factory that uses a template method (`handle_event`) to create handlers for communication channels. The `handle_event` method implements the algorithm that passively listens for connection requests, then creates and activates a handler when the connection is established. The handler performs a service using data exchanged on the connection. Thus, the service is decoupled from the network programming interface and the transport protocol used to establish the connection.

The Router pattern is a specialization of the Gateway pattern in [6]. The Gateway pattern decouples cooperating components of a software system and allows them to interact without having direct dependencies among each other. The Router pattern decouples the mechanisms used to process input messages from the mechanisms used to process output mechanisms to prevent blocking. In addition, this pattern allows the use of different concurrency strategies for input and output channels.

6 Concluding Remarks

This paper describes a system of design patterns and framework components used to build high-performance communication `Gateways`. The design patterns presented in this paper capture the collaboration between framework components that perform common communication software tasks (such as event demultiplexing, event handler dispatching, connection establishment, routing, configuration of application services, and concurrency control). The family of design patterns and the ACE framework components described in this paper have been reused by the author and his colleagues in a number of production communication software systems.

In general, our experience applying reuse strategies based on design patterns and frameworks has been positive. For instance, the ability to document the intent, structure, and behavior of components in the ACE framework in terms of patterns has significantly reduced software development effort for projects where it has been applied. An in-depth discussion of our experiences and lessons learned using patterns appeared in [2].

The object-oriented ACE components described in this paper are freely available via the WWW at `http://www.cs.wustl.edu/~schmidt/ACE.html`. This distribution contains complete source code, documentation, and example test drivers for the C++ components developed as part of the ADAPTIVE project [4] at the University of California, Irvine and Washington University.

References

[1] 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.

[2] 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.

[3] R. Johnson and B. Foote, "Designing Reusable Classes," *Journal of Object-Oriented Programming*, vol. 1, pp. 22–35, June/July 1988.

[4] D. C. Schmidt, "ACE: an Object-Oriented Framework for Developing Distributed Applications," in *Proceedings of the* 6^{th} *USENIX C++ Technical Conference*, (Cambridge, Massachusetts), USENIX Association, April 1994.

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

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

[7] G. Booch, *Object Oriented Analysis and Design with Applications (*2^{nd} *Edition)*. Redwood City, California: Benjamin/Cummings, 1993.

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

[9] D. C. Schmidt, "Connector: a Design Pattern for Actively Initializing Network Services," *C++ Report*, vol. 8, January 1996.

[10] D. C. Schmidt, "A Family of Design Patterns For Flexibly Configuring Network Services in Distributed Systems," in *International Conference on Configurable Distributed Systems*, May 6–8 1996.

[11] D. C. Schmidt, T. H. Harrison, and E. Al-Shaer, "Object-Oriented Components for High-speed Network Programming," in *Proceedings of the* 1^{st} *Conference on Object-Oriented Technologies and Systems*, (Monterey, CA), USENIX, June 1995.

[12] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, 1.2 ed., 1993.

[13] D. C. Schmidt, T. H. Harrison, and I. Pyarali, "Experience Developing an Object-Oriented Framework for High-Performance Electronic Medical Imaging using CORBA and C++," in *Proceedings of the "Software Technology Applied to Imaging and Multimedia Applications mini-conference" at the Symposium on Electronic Imaging in the International Symposia Photonics West*, SPIE, January 1996.

[14] Object Management Group, *Universal Networked Objects*, TC Document 95-3-xx ed., Mar. 1995.

[15] J. O. Coplien, "A Development Process Generative Pattern Language," in *Pattern Languages of Programs* (J. O. Coplien and D. C. Schmidt, eds.), Reading, MA: Addison-Wesley, June 1995.

[16] D. C. Schmidt, "Acceptor and Connector: Design Patterns for Actively and Passively Initializing Network Services," in *Workshop on Pattern Languages of Object-Oriented Programs at ECOOP '95*, (Aarhus, Denmark), August 1995.

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

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

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

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

[21] R. G. Lavender and D. C. Schmidt, "Active Object: an Object Behavioral Pattern for Concurrent Programming," in *Proceedings of the* 2^{nd} *Annual Conference on the Pattern Languages of Programs*, (Monticello, Illinois), pp. 1–7, September 1995.

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

[23] S. Berczuk, "A Pattern for Separating Assembly and Processing," in *Pattern Languages of Program Design* (J. O. Coplien and D. C. Schmidt, eds.), Reading, MA: Addison-Wesley, 1995.