A Family of Design Patterns for Application-Level Gateways

Douglas C. Schmidt
schmidt@cs.wustl.edu
http://www.cs.wustl.edu/∼schmidt/
Department of Computer Science
Washington University, St. Louis 63130
(TEL) 314-935-7538, (FAX) 314-935-7302

This paper 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

Abstract

Developers of communication software must confront recurring design challenges involving robustness, efficiency, and extensibility. Many of these challenges are independent of the application-specific requirements. Successful developers resolve these challenges by applying appropriate design patterns. However, these patterns have traditionally been locked in the minds of expert developers or buried within complex system source code. The primary contribution of this paper is to describe a family of design patterns that underly many object-oriented communication software systems. In addition to describing each pattern separately, the paper illustrates how knowledge of the relationships and trade-offs among patterns helps guide the construction of reusable communication software frameworks.

1 Introduction

Building, maintaining, and enhancing high quality communication systems is hard. Developers must have a deep understanding of many complex issues such as service initialization and distribution, concurrency control, flow control, error handing, and event loop integration. Successful communication software created by experienced software developers embodies solutions to these issues.

It is often difficult, however, to separate the essence of successful software solutions from the details of a particular implementation. Even when software is written using well-structured object-oriented frameworks and components, it can be hard to identify key roles and relationships. Moreover, OS platform features (such as the absence or presence of multi-threading) or 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 and articulating the essence and commonality of successful communication software is important for several reasons:

- *It helps guide the design choices of developers who are building new communication systems* – By understanding the potential traps and pitfalls in their domain, developers can select suitable architectures, protocols, and platform features without wasting time and effort implementing inefficient or error-prone solutions.
- *It preserves important design information for programmers who enhance and maintain existing software* – Often, this information is locked in the minds of the original developers. If this design information is not documented explicitly, however, it will be lost over time, thereby increasing maintenance costs and decreasing software quality.

The purpose of this paper is to illustrate an effective way to document the essence of successful communication software by describing key *design patterns* used to build application-level Gateways, which route messages between Peers distributed throughout a communication system.

Design patterns represent successful solutions to problems that arise when building software [1]. Capturing and articulating key design patterns helps 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 costly trial and error.

This paper presents the object-oriented architecture and design of an application-level `Gateway` in terms of the design patterns used to guide the construction of reusable and `Gateway`-specific frameworks and components. Application-level Gateways have stringent requirements for reliability, performance, and extensibility. Therefore, they are excellent exemplars for presenting the structure, participants, and consequences of key design patterns that appear in many communication software systems.

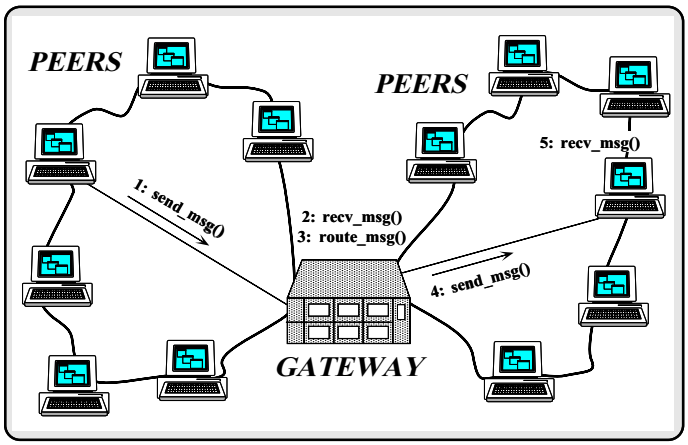

Figure 1: The Structure and Collaboration of Peers and the Gateway

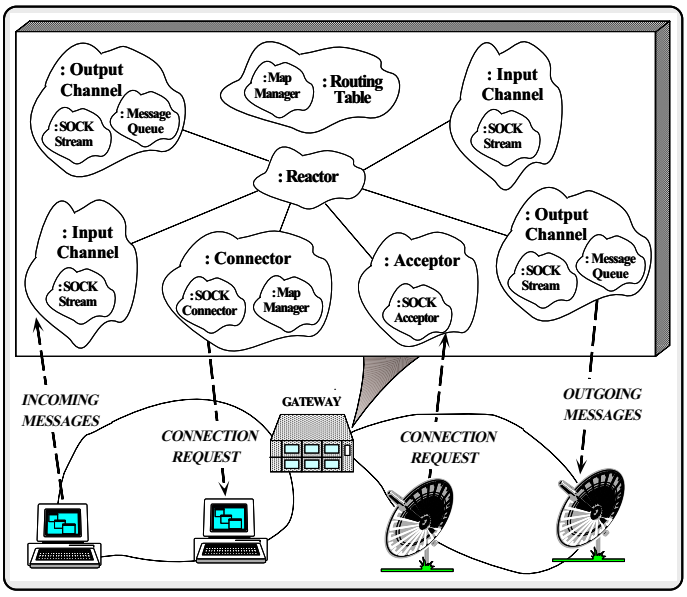

Figure 2: The Object-Oriented `Gateway` Software Architecture

The patterns described in this paper were discovered while building a wide range of communication systems including on-line transaction processing systems, telecommunication switch management systems [2], electronic medical imaging systems [3], and parallel communication subsystems [4]. Although the specific application requirements in these systems were quite different, the communication software design challenges were very similar. Therefore, although the examples in this paper focus on Gateways, the patterns embody design expertise that can be reused broadly in the communication domain.

The remainder of this paper is organized as follows: Section 2 outlines an object-oriented software architecture for application-level Gateways; Section 3 examines the design patterns that form the basis for reusable communication software, using application-level Gateways as an example; Section 4 compares these patterns with those described in related work; and Section 5 presents concluding remarks.

2 An Object-Oriented Software Architecture for Application-level Gateways

This paper examines framework components and design patterns that comprise and motivate the object-oriented architecture of application-level Gateways developed by the author and his colleagues. A Gateway is a Mediator [1] that decouples cooperating Peers throughout a network and allows them to interact without having direct dependencies on each other [5]. 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 collaborations among objects within a software architecture for application-level Gateways.[1] This architecture is based on extensive experience developing connection-oriented Gateways for various commercial and research communication systems. After building multiple Gateway systems it became clear that the software architecture of these systems was largely independent of the protocols used to route messages to Peers. This realization enabled the components depicted in Figure 2 to be reused for the communication software subsystems of many other projects. The ability to reuse these components so widely stems from two factors:

• **Understanding the actions and interactions of key design patterns within the domain of communication software:** Patterns capture the structure and collaboration of participants in a software architecture at a higher level than source code and object-oriented design models that focus on individual objects and classes. Some of the communication software patterns described in this paper have been documented individually [7, 8, 9]. 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 *family of design patterns* for communication software. These design patterns work together to solve complex problems within the domain of communication software.

[1] Relationships between components are illustrated throughout this paper using Booch notation [6]. In this figure solid clouds indicate objects; nesting indicates composition relationships between objects; and undirected edges indicate an association exists between two objects.

• **Developing an object-oriented framework that implements these design 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 with the ADAPTIVE Communication Environment (ACE) software [10]. ACE provides an integrated framework of reusable C++ wrappers 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 design patterns described in Section 3. However, the patterns are much richer than their implementation in ACE 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 family of design patterns that underly the ACE components.

2.1 Application-independent Components

Most of the 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 Input and Output Channels, which implement the application-specific details related to message formats and the 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 [11]. These components simplify the development of portable and correct communication software by shielding developers from low-level, tedious, and error-prone socket-level programming. In addition, they form the foundation for the higher-level ACE components and patterns described below.

• **Service initialization components:** The Connector and Acceptor are factories [1] that implement active and passive strategies for initializing network services, respectively.[2] These components are based on the Connector pattern described in Section 3.2 and Acceptor pattern described in Section 3.3. The Gateway uses these components to establish connections with Peers and produce initialized Input and Output Channels.

[2]Establishing connections between endpoints involves two roles: 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 the *active role* (which actively initiates a connection to one or more endpoints that are playing the passive role).

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

1. *From the* Gateway *to the* Peers – which is typically done whenever 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 whenever a new Peer wants to send or receive routing messages.

In a large system, several scores of Peers may be connected to a single Gateway. Therefore, to expedite initialization, 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 [11] 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.

• **Event demultiplexing components:** The Reactor is an object-oriented event demultiplexing mechanism based on the Reactor pattern [7] described in Section 3.1. It channels all external stimuli in an event-driven application to a single demultiplexing point. This 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 is routing messages that encapsulate various payloads (such as commands, status messages, and bulk data transmissions).

• **Message demultiplexing components:** The Map Manager is a parameterized collection that efficiently maps external ids (*e.g.*, Peer routing addresses) onto internal ids (*e.g.*, Output Channels). 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 Output Channels.

• **Message queueing components:** The Message Queue [10] provides a generic queueing mechanism. This mechanism runs efficient and robustly in multi-threaded or single-threaded environments. Developers can select the desired concurrency strategy at the time a queue is instantiated, The Gateway uses Message Queues to buffer messages in Output Channels while they are being routed to Peers.

2.2 Application-specific Components

Only two of the components (Input and Output Channels) in Figure 2 are specific to the Gateway application. These components implement the Router pattern

described in Section 3.4. `Input` and `Output Channels` 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 described below:

• **Input Channels:** `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 connection's communication 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 requests the selected `Output Channels` to forward the message to the appropriate `Peer` destinations.

• **Output Channels:** 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 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 `Output Channel` to stop sending messages until the destination `Peer` consumes its existing data.

A `Gateway` integrates the application-specific and application-independent components by inheriting from, instantiating, and composing the ACE components described above. As shown in Figure 3[3] `Input` and `Output Channels` inherit from a common ancestor: the ACE `Svc Handler` class, which is produced by `Connectors` and `Acceptors`. The `Svc Handler` is a local Proxy [1] for a remotely connected `Peer`. It provides a `SOCK Stream`, which enables `Peers` to exchange messages via connected `Channels`.

An `Output Channel` uses an ACE `Message Queue` to chain unsent messages in the order they must be delivered when flow control mechanisms permit. Once a flow controlled connection opens up, the ACE framework notifies its `Output Channel`, 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 flexibility in connection configuration and concurrency strategies;
- It enhances reliability if errors occur on a connection since `Input` and `Output Channels` can be reconnected independently.

[3] This figure illustrates additional Booch notation. 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 relation between two classes.

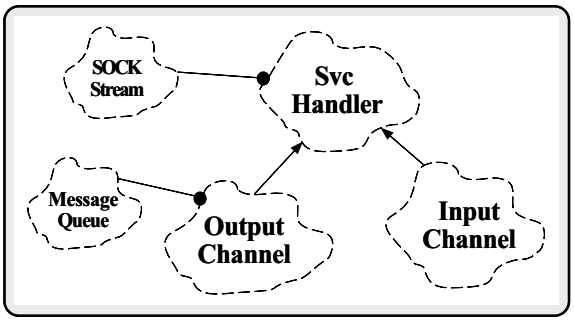

Figure 3: Channel Inheritance Hierarchy

3 A Family of Design Patterns for Application-level Gateways

Section 2 illustrates 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 the various 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 *design patterns* that reify them. 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). Studying the patterns in `Gateway` software is important to:

1. *Identify successful solutions to common design challenges* – The patterns underlying the `Gateway` architecture transcend the particular details of the application and resolve common challenges faced by communication software developers. A thorough understanding of the patterns presented below enables widespread reuse of `Gateway` software architecture in other systems, even when reuse of its algorithms, implementations, interfaces, or detailed designs is not feasible [12].

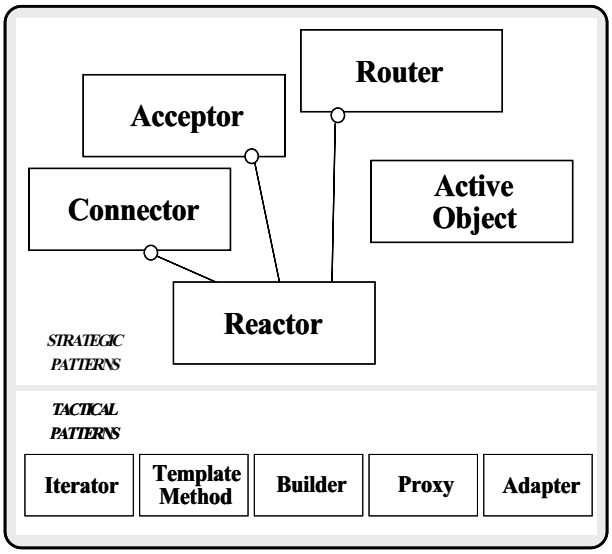

Figure 4: The Family of Patterns for Application-level `Gateways`

2. *Reduce the effort of maintaining and enhancing* `Gateway` *software* – The use of patterns 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.

Figure 4 illustrates the following *strategic* patterns related to connection-oriented, application-level `Gateways`:

- **Concurrency patterns:**
  - *The Reactor pattern* – which decouples event demultiplexing and event handler dispatching from services performed in response to events;
  - *The Active Object pattern* – which decouples method execution from method invocation.
- **Service initialization patterns:**
  - *The Connector pattern* – which decouples active service initialization from the tasks service performed once the service is initialized;
  - *The Acceptor pattern* – which decouples passive service initialization from the tasks performed once the service is initialized.
- **Application-specific patterns:**
  - *The Router pattern* – which decouples input mechanisms from output mechanisms to route data correctly without blocking a `Gateway`.

These five patterns are strategic because they significantly influence the software architecture for applications in a particular domain (in this case, the domain of communication software and `Gateways`). For example, the Router pattern described in Section 3.4 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 helps to ensure a consistently high quality of service for `Gateways` that use reliable 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 application-level Gateways.

Application-level Gateways also utilize many *tactical* patterns, such as the following:

- *Builder pattern* – which provides a factory for building complex objects incrementally. The `Gateway` uses this pattern to create its `Routing Table` from a configuration file.
- *Iterator pattern* – which decouples sequential access to a container from the representation of the container. The `Gateway` uses this pattern to connect and initialize `Input` and `Output Channels` with `Peers`.
- *Template Method pattern* – which specifies an algorithm where some steps are supplied by a derived class. The `Gateway` uses this pattern to selectively override certain steps in the `Connector` and `Acceptor` in order to restart failed connections automatically.
- *Adapter pattern* – which 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) in a uniform manner.
- *Proxy pattern* – which 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.

Compared to strategic patterns (which are often domain-specific and have broad design implications), 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 like the Router. A thorough understanding of tactical patterns is essential to implement highly flexible software that is resilient to changes in application requirements and platform environments.

Although there are various forms for describing patterns, they typically convey the following information [1]:

- The intent of the pattern;

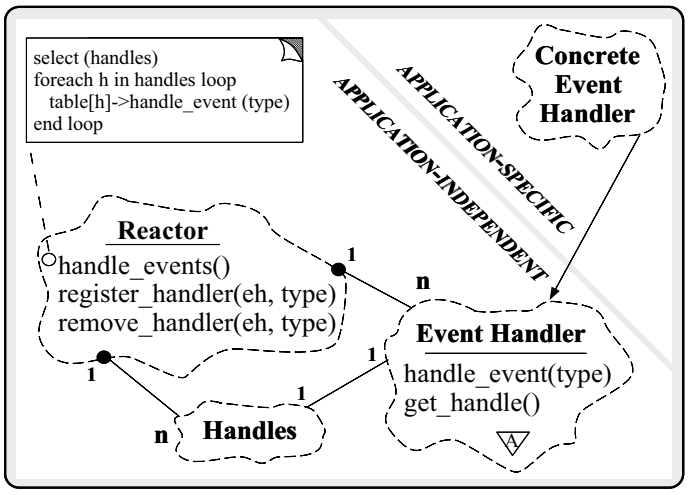

Figure 5: Structure and Participants in the Reactor Pattern

- The design forces that motivate and shape 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.

It is important to recognize that the strategic patterns in this paper are much more generally applicable than the specific use cases for the `Gateway` described below. The references [7, 8, 9] provide additional use cases for these patterns, along with more detailed coverage of each pattern and sample implementations.

3.1 The Reactor Pattern

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

Motivation and Forces: Single-threaded applications must be able to handle events from multiple sources without blocking on any single source. The Reactor pattern resolves the following forces that 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* – A Reactor serializes the handling of events from multiple sources within an application 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 from the event handler processing policies. The demultiplexing and dispatching mechanisms are generally independent of an application and are thus reusable. In contrast, the event handler policies are more specific to an application. This separation of concerns allows application policies to change without affecting the lower-level framework mechanisms.

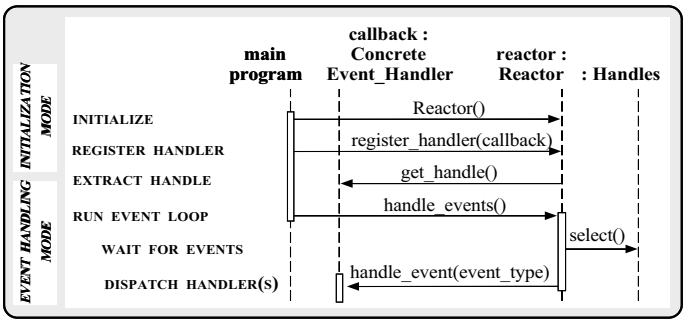

Figure 6: Object Interaction Diagram for the Reactor Pattern

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 `Input` or `Output Channels` in the `Gateway`). 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 (such as input, output, signal, and timer events).

An `Event Handler` specifies an abstract interface used by the `Reactor` to dispatch callback methods defined by objects that register to 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 modes:

1. *Initialization mode* – where `Concrete Event Handler` objects are registered with the `Reactor`;
2. *Event handling mode* – where the `Reactor` invokes upcalls on registered objects, which then handle events in an application-specific way.

Usage: The `Reactor` is used for the following event dispatching operations in a `Gateway`:

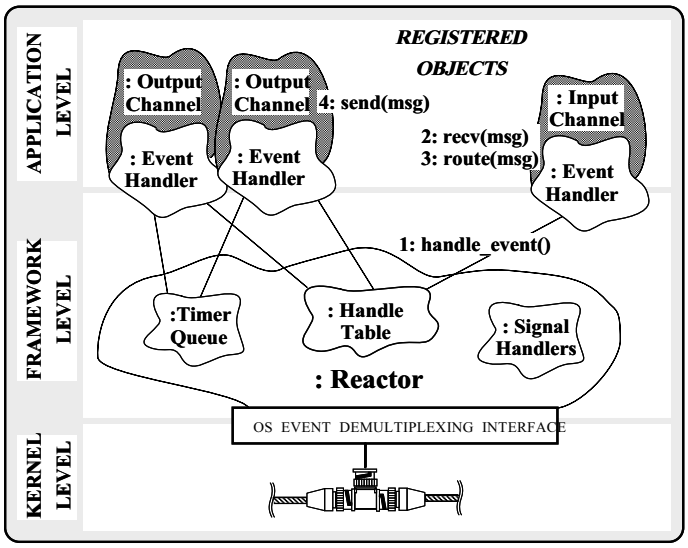

Figure 7: Using the Reactor Pattern in the `Gateway`

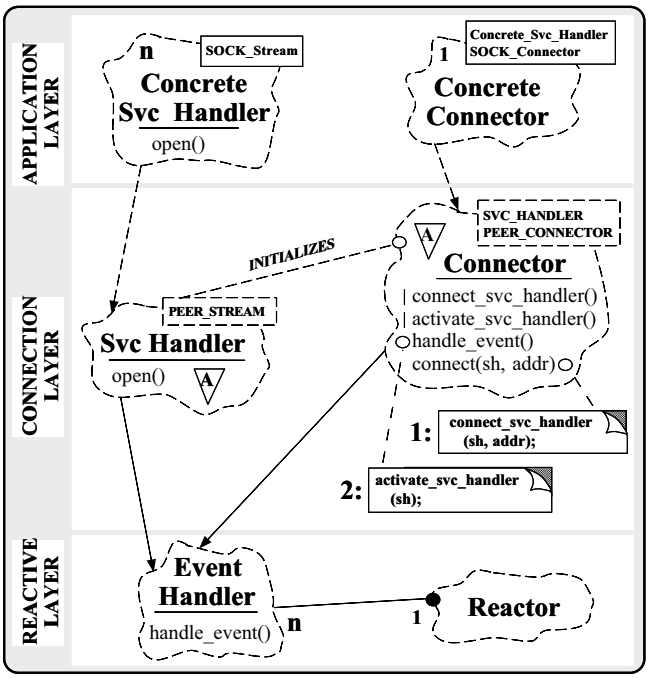

Figure 8: Structure and Participants in the Connector Pattern

- *Input events* – The `Reactor` dispatches each incoming routing message to the `Input Channel` associated with its socket handle, at which point the message is routed to the appropriate `Output Channel`(s). This use-case is shown in Figure 7.
- *Output events* – The `Reactor` ensures that outgoing routing messages are eventually delivered on flow controlled `Output Channels` described in Section 3.4 and 3.5.
- *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` described in Section 3.2.
- *Connection request events* – The `Reactor` also dispatches events that indicate the arrival of passively initiated connections. These events are used by the `Acceptor` described in Section 3.3.

The Reactor pattern has been used in many single-threaded event-driven frameworks (such as the Motif, Interviews [13], System V STREAMS [14], the ACE object-oriented communication framework [10], and implementations of DCE and CORBA). In addition, it forms the foundation for most of the strategic patterns presented below.

3.2 The Connector Pattern

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

Motivation and Forces: Connection-oriented applications (like a `Gateway`) and middleware (like CORBA or Distributed COM) are often written using lower-level network programming interfaces (like sockets [15] and TLI [16]). The Connector pattern resolves the following forces that impact the active initialization of services written using these lower-level interfaces:

1. *The need to reuse active connection establishment code for each new service* – The Connector pattern permits key characteristics of services (such as the communication protocol 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 reduces 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 like sockets and TLI.
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 or how the services were initialized.
4. *The need to actively establish connections with large number of peers efficiently* – The Connector pattern can

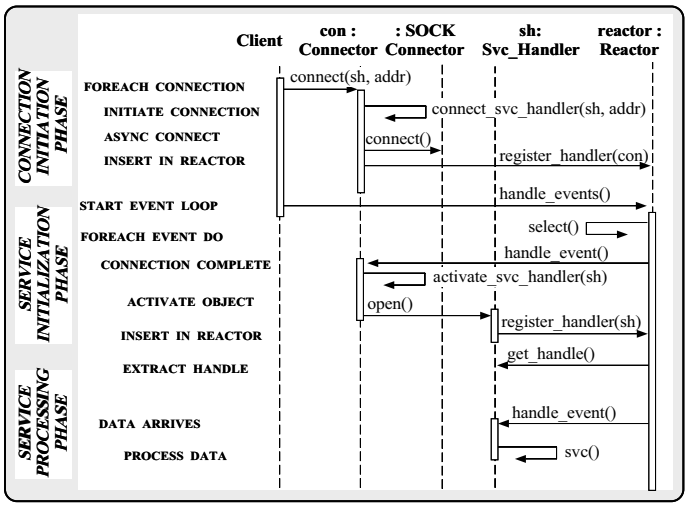

Figure 9: Object Interaction Diagram for the Connector Pattern

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.

Structure, Participants, and Implementation: Figure 8 illustrates the layering structure of participants in the Connector pattern.[4] The `Connector` is a factory that assembles the resources necessary to connect and activate a `Svc Handler`, which is a Proxy that exchanges messages with its `Peer`. The `Connector`'s initialization strategy can establish connections with `Peers` either synchronously or asynchronously.

The participants in the Connection Layer of the 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 actively within a single thread of control.

To increase flexibility, the implementation of a `Connector` can be 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 by the `Connector` to actively 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 `Connector` associates this endpoint to its `Peer` when a connection is actively established.

[4] In this figure the dashed rectangles indicate template parameters and a dashed directed edge indicates template instantiation.

By inheriting from `Event Handler` (shown in Figure 5), 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 `Connector`. Conversely, a `Svc Handler` can use the Active Object pattern and handle its I/O events within a separate thread. Section 3.5 evaluates the tradeoffs between these different patterns.

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 supply template arguments for these types to produce Application Layer `Connectors` (such as the `Connector` used by the `Gateway` to initialize its `Input` and `Output Channels`). This enables the wholesale replacement of the `SVC HANDLER` and `PEER CONNECTOR` types, without affecting the 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 [1]. 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.

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 be initiated synchronously or asynchronously. The `Connector`'s `connect` method implements the strategy for actively establishing connections.
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 9 illustrates these three phases of collaboration using *asynchronous* connection establishment. Note how the connection initiation phase is temporally separated from the service initialization phase. This enables multiple connection initiations to proceed in parallel within a single thread of control. The collaboration for synchronous connection establishment is similar. In this case, the `Connector` 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 (*e.g.*, 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 like the `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 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 a `Connector` and a `Svc Handler` occurs in the service initialization phase, when the `Connector` invokes the `open` method of the `Svc Handler`. 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 `Input Channels` to frame the messages, determine outgoing routes, and deliver the messages to their `Output Channels`. However, a different type of concurrency mechanism (such as Active Objects described in Section 3.5) can be used by the `Output Channels` to send the data to the remote destinations.

Usage: The Connector pattern is used by the `Gateway` to simplify the task of connecting to a large number of `Peers`. `Peer` addresses are read from a configuration file during `Gateway` initialization. The `Gateway` uses the Builder pattern [1] to bind these addresses to dynamically allocated `Channels`. Since `Channels` inherit from `Svc Handler`, all connections can be initiated asynchronously using the Iterator pattern [1]. The connections are then completed in parallel using the `Connector`.

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

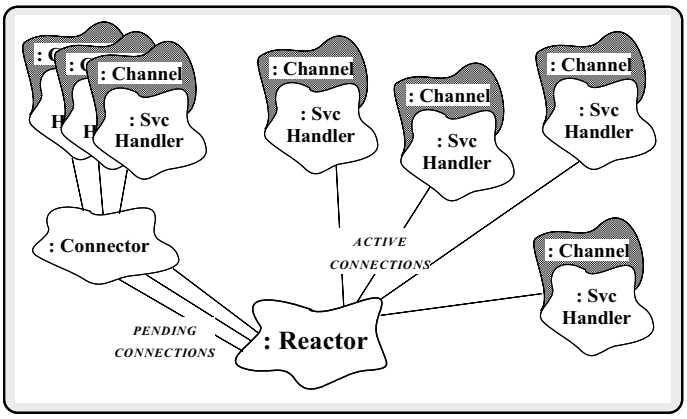

Figure 10: Using the Connector Pattern in the `Gateway`

connections complete, the `Connector` removes each connected `Channel` from its table and activates it. In the single-threaded implementation `Input Channels` register themselves with the `Reactor` once they are activated. 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 3.4).

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 an `Output Channel` due to network congestion). If a connection fails unexpectedly, an exponential-backoff algorithm can be implemented using the timer-based dispatching capabilities of the `Reactor` to restart the connection efficiently.

3.3 The Acceptor Pattern

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

Motivation and Forces: The Acceptor pattern resolves the following forces that impact the passive initialization of services written using lower-level network programming interfaces like sockets and TLI:

1. *The need to reuse passive connection establishment code for each new service* – The Acceptor pattern permits key characteristics of services (such as the communication protocol 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.

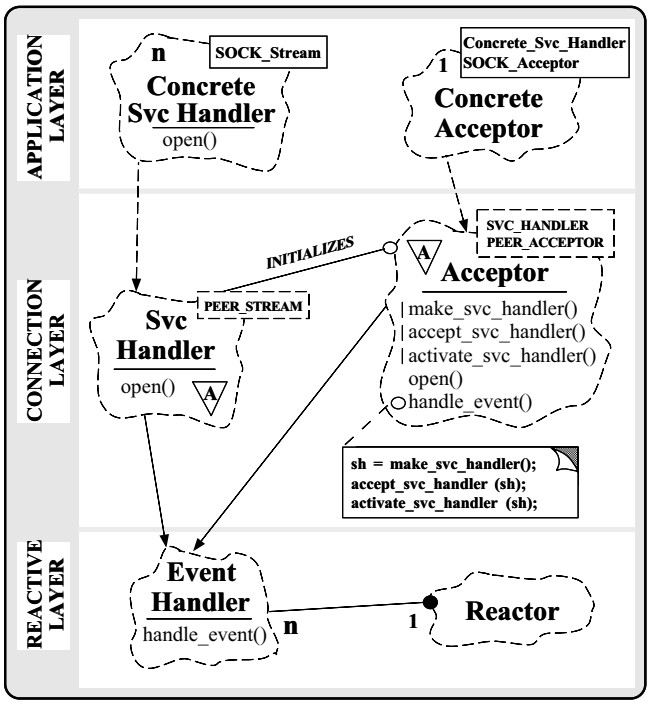

Figure 11: Structure and Participants in the Acceptor Pattern

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 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 `Acceptor` from the `Svc Handler`, passive-mode listener endpoints cannot be used incorrectly (*e.g.*, to try to read or write data on a passive-mode listener socket used to accept connections). This eliminates an important class of network programming errors.

The Acceptor pattern is the "dual" of the Connector pattern described in Section 3.2. However, the Connector pattern establishes connections *actively*, whereas the Acceptor pattern establishes connections *passively*. The consequences of this

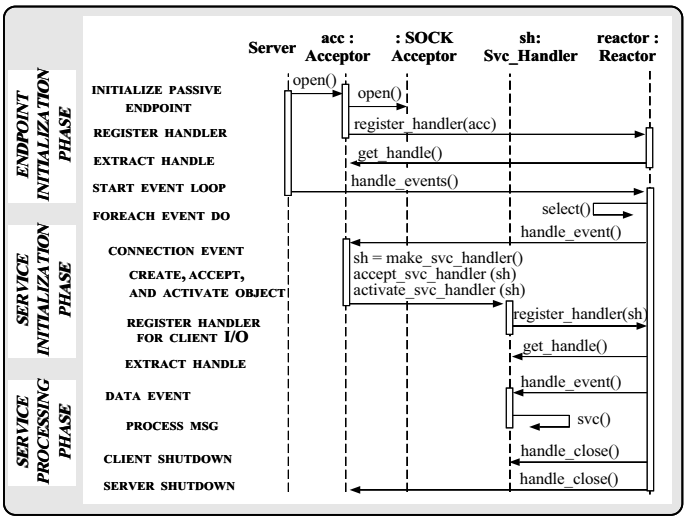

Figure 12: Object Interaction Diagram for the Acceptor Pattern

difference in connection roles is illustrated by the forces these two patterns resolve. For instance, note that the first three forces resolved by the Acceptor pattern are essentially the same as for the Connector pattern, with only the passive and active roles reversed. However, the final force resolved in each pattern is different due to the inverse connection roles played by each pattern.

Structure, Participants, and Implementation: Figure 11 illustrates the layering structure of participants in the Acceptor pattern, which is nearly identical to the Connector layering structure in Figure 8. The `Acceptor` is a factory that assembles the resources necessary to create, accept, and activate a `Svc Handler`. The `Svc Handler` in the Acceptor pattern plays the same role as in the Connector pattern, *i.e.*, it is a local Proxy for a remotely connected `Peer`.

The Connection Layer in the Acceptor pattern leverages off the Reactor pattern. For instance, the `Acceptor`'s initialization strategy establishes a connection after the `Reactor` notifies it that a new connection request has arrived from a `Peer`. Using the Reactor pattern enables multiple `Svc Handlers` to be initialized passively within a single thread of control.

To increase flexibility, the implementation of an `Acceptor` can be parameterized by a particular type of `PEER CONNECTOR` and `SVC HANDLER`. The `PEER ACCEPTOR` supplies the underlying transport mechanism (such as C++ wrappers for sockets or TLI) used by the `Acceptor` to passively 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` associates this endpoint to its `Peer` when a connection is established passively.

As with the Connector pattern, a `Svc Handler` can use either the Reactor pattern or Active Object pattern to handle

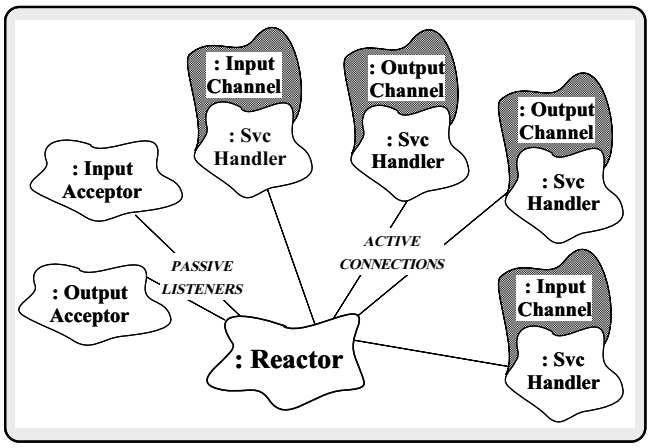

Figure 13: Using the Acceptor Pattern in the `Gateway`

its I/O events. Likewise, the implementation of the Acceptor pattern presented above also uses parameterized types. Parameterized types enhance portability since the Acceptor pattern's connection establishment strategy is independent of the type of service and the type of IPC mechanism. Developers can supply concrete arguments for these types to produce Application Layer `Concrete Acceptor` (such as the `Acceptor` used by the `Gateway` and `Peers` to passively initialize `Input` and `Output Channels`).

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 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 events arrive. This method performs the `Acceptor`'s `Svc Handler` initialization 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 a concurrent event handling pattern such as the Reactor or the Active Object [9].

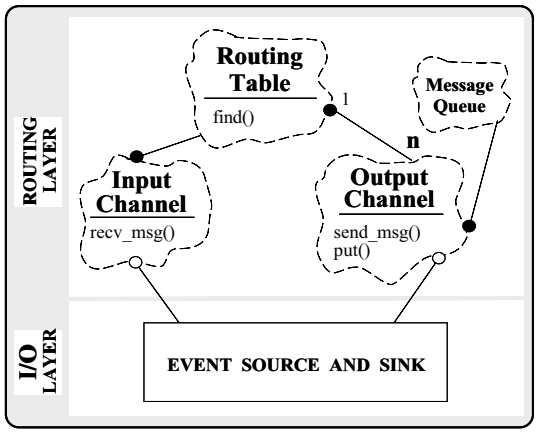

Figure 14: Structure and Participants in the Router Pattern

Usage: 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, `Peers` connect to `Gateway`, which uses the Acceptor pattern to decouple the passive initialization of `Input` and `Output Channels` from the routing tasks performed once a `Channel` is initialized.

The intent and general architecture of the Acceptor pattern is found in network server management tools like `inetd` [15] and `listen` [16]. 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.

3.4 The Router Pattern

Intent: The Router pattern decouples multiple sources of input from multiple sources of output to route data correctly without blocking a `Gateway`.

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 links. The Router pattern resolves the following forces that arise when building robust connection-oriented `Gateways`:

1. *The need to prevent misbehaving connections from disrupting the quality of service for well-behaved connections* – If outgoing connections can flow control as a result of network congestion, or input connections can fail because `Peers` disconnect, the `Gateway` must not perform blocking `send` or `recv` operations on any single channel. Otherwise, messages on other channels could not be sent or received and the quality of service provided to `Peers` would degrade.
2. *The need to allow different concurrency strategies for Input and Output Channels* – Several concur-

rency strategies for processing `Input` and `Output Channels` are described in this paper including (1) single-threaded processing using the Reactor pattern and (2) multi-threaded processing using the Active Object pattern. Each strategy is 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 customized concurrency strategies to be configured flexibly into a `Gateway`.

Structure, Participants, and Implementation: Figure 14 illustrates the layer structuring of participants in the Router pattern. The I/O Layer provides an event source for `Input Channels` and an event sink for `Output Channels`. An `Input Channel` uses a `Routing Table` to map routing messages onto one or more `Output Channels`. If messages cannot be delivered to their destination `Peers` immediately they are buffered in a `Message Queue` for subsequent transmission.

Because `Input Channels` are decoupled from `Output Channels` their implementations can vary independently. This separation of concerns is important since it allows different concurrency strategies to be used for input and output. The consequences of this decoupling is discussed further in Section 3.5.

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

1. *Input processing phase* – where `Input Channels` reassemble incoming TCP segments into complete routing messages;
2. *Route selection phase* – where `Input Channels` consult a `Routing Table` to select the `Output Channels` responsible for sending the routing messages;
3. *Output processing phase* – where the selected `Output Channels` transmit the routing messages to their destination(s) *without* blocking the process.

Usage: The other strategic patterns in this paper (*i.e.*, Reactor, Connector, Acceptor, and Active Object) can be applied to many other types of communication software. In contrast, the Router pattern is tightly coupled with the `Gateway` application. A primary challenge of building a reliable connection-oriented `Gateway` centers on avoiding blocking I/O. This is necessary to reliably manage flow control on `Output Channels`. 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 `Output Channels`.

The remainder of this section describes how the Router pattern can be implemented in a single-threaded, Reactor version of the `Gateway` (Section 3.5 examines the multi-threaded,

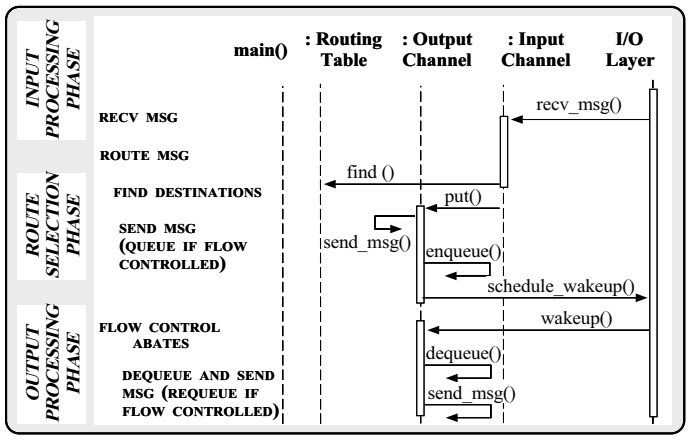

Figure 15: Object Interaction Diagram for the Router Pattern

Active Object version of the Router pattern). The Router pattern uses a `Reactor` as a cooperative multi-tasking scheduler for `Gateway` I/O operations, just like the Connector and Acceptor patterns. The `Reactor` allows multiple events on different connections to be demultiplexed within a single thread of control. 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).

In the Reactor version of the Router pattern, the `Input Channels` and `Output Channels` inherit indirectly from `Event Handler`. This enables the `Gateway` to route messages by having the `Reactor` dispatch the `handle_event` methods of `Input` and `Output Channels` when messages arrive and flow control conditions subside, respectively.

Using the Reactor pattern to implement the Router pattern involves the following steps:

1. *Initialize non-blocking endpoints* – The `Input` and `Output Channel` 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 subtle errors that can occur on faulty or congested network links.
2. *Input message reassembly and routing* – Routing messages are received in fragments by `Input Channels`. If an entire message is not immediately available, the `Input Channel` must buffer the fragment and return control to the event loop. This is essential to prevent "head of line" blocking on `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` that will deliver the message.
3. *Message delivery* – The selected `Output Channels` try to send the message to the destination `Peer`. Mes-

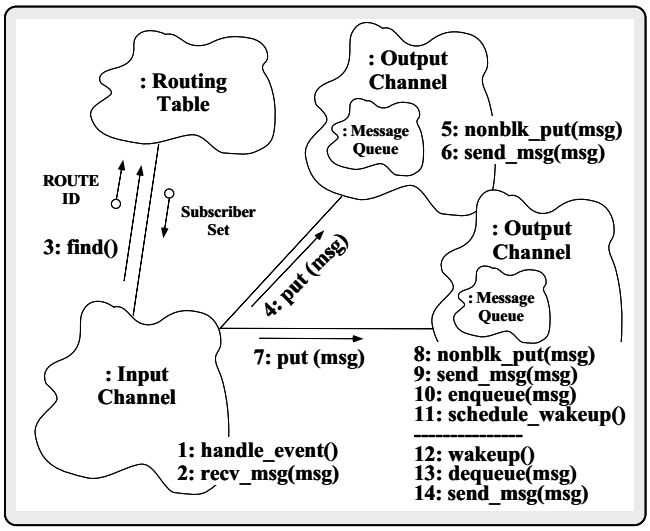

Figure 16: Using the Router Pattern in a Single-threaded Reactive `Gateway`

sages must be delivered reliably in "first-in, first-out" (FIFO) order. To avoid blocking, all `send` operations in `Output Channels` 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 `Output Channel` in the upper right-hand corner of Figure 16. If the link *is* flow controlled, however, the Router pattern must use a different strategy. This path is depicted by the `Output Channel` in the lower right-hand corner of Figure 16.

To handle flow controlled connections, 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 correctly routes the messages to peers without blocking on any single I/O channel.

3.5 The Active Object Pattern

Intent: The Active Object pattern decouples method execution from method invocation to enable concurrent execution of methods.

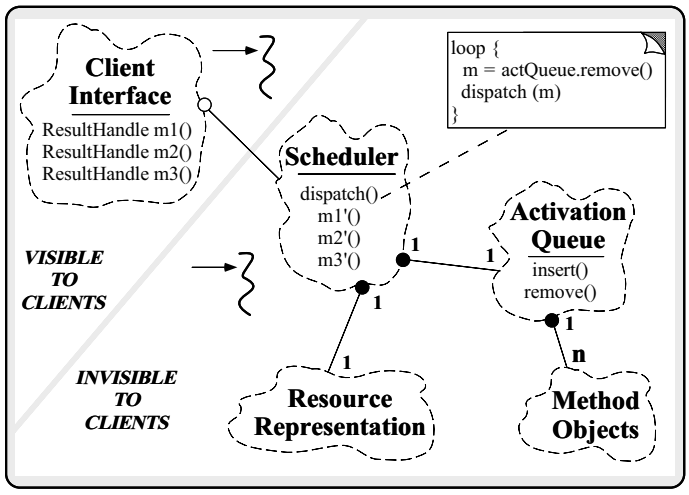

Figure 17: Structure and Participants in the Active Object Pattern

Motivation and Forces: All the strategic patterns used by the single-threaded `Gateway` in Section 3.4 are layered upon the Reactor pattern. The Connector, Acceptor, and Router patterns all 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 `Output Channels` during periods of network congestion). In fact, much of the complexity in the single-threaded Router pattern implementation stems from using the Reactor pattern as a cooperative multi-tasking mechanism. It is much easier, therefore, to implement the output portion of the Router pattern with the *Active Object* pattern. This pattern allows `Output Channels` to block independently when sending messages to `Peers`.

The Active Object pattern resolves the following force that impacts the design of applications like a `Gateway` that must communicate simultaneously with multiple `Peers`:

- *The need to allow blocking read and write operations on one endpoint that do not detract from the quality of service of other endpoints.* Network services are generally easier to program if blocking I/O is used rather than reactive non-blocking I/O [17]. The increased simplicity occurs since the execution state can be localized in the activation records of a thread of control, rather than being decentralized in a set of control blocks maintained

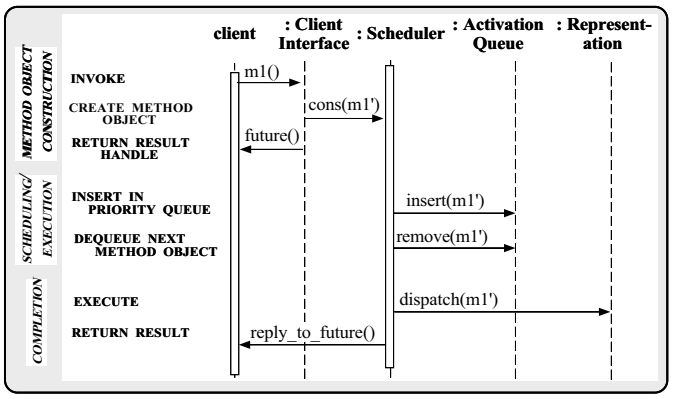

Figure 18: Object Interaction Diagram for the Active Object Pattern

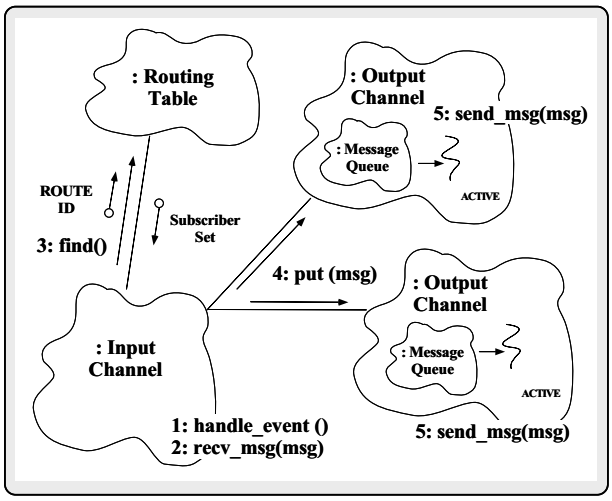

Figure 19: Using the Router Pattern in a Multi-threaded Active Object `Gateway`

by application developers.

Structure, Participants, and Implementation: Figure 17 illustrates the structure and participants in the Active Object pattern. The `Client Interface` presents the public methods available to clients. The `Scheduler` determines next method to execute based on synchronization and scheduling constraints. The `Activation Queue` maintains a list of pending `Method Objects`. The `Scheduler` determines the order in which these `Methods Objects` are executed (a FIFO scheduler is used in the `Gateway` to maintain the order of message delivery). The `Resource Representation` maintains context information shared by the implementation methods.

Collaborations: Figure 18 illustrates the collaborations among participants in the Active Object pattern. These collaborations are divided into the following phases:

1. *Method Object construction* – in this phase the client application invokes a method defined by the `Client Interface`. This triggers the creation of a `Method Object`, which maintains the argument bindings to the method, as well as any other bindings required to execute the method and return a result. For example, a binding to a `Result Handle` object returned to the caller of the method. A `Result Handle` is returned to the client unless the method is "oneway," in which case no `Result Handle` is returned.
2. *Scheduling/execution* – in this phase the `Scheduler` acquires a mutual exclusion lock, consults the `Activation Queue` to determine which `Method Object(s)` meet the synchronization constraints. The `Method Object` is then bound to the current `Resource Representation` and the method is allowed to access/update this representation and create a `Result Handle`.
3. *Return result* – the final phase binds the `Result Handle` value, if any, to a *future* [18, 19] object that 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 Object` will be garbage collected when they are no longer needed.

Usage: The `Gateway` implementation described in Section 3.4 is single-threaded. It uses the Reactor pattern implementation of the Router 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, maintenance programmers frequently did not recognize the importance of returning control to the `Reactor`'s event loop immediately 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. The remainder of this section describes how `Output Channels` can be multi-threading using the Active Object pattern.[5] This modification greatly simplified the implementation of the Router pattern since `Output Channels` can block in their own thread of control without affecting other `Channels`. Implementing the `Output Channels` as Active Objects also eliminated the subtle and error-prone cooperative multi-tasking programming techniques required when using the `Reactor` to schedule `Output Channels`.

Figure 19 illustrates the Active Object version of the Router pattern. Note how much simpler is it compared with

[5]While it is possible to apply the Active Object pattern to the `Input Channels` this has less impact on the `Gateway` design because the `Reactor` already supports non-blocking input.

the Reactor solution in Figure 16. The simplification occurs primarily since the complex output scheduling logic moved into the Active Objects, rather than being the responsibility of the application programmer.

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 Router pattern in production `Gateway` systems. [6] However, using source code to identify the reasons behind this complexity is hard due to all the error handling and protocol-specific details that surround 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 vs. the Active Object pattern.* This paper has explicitly focused on the interactions and tradeoffs between these patterns in order to clarify the consequences of different design choices. In general, documenting the interactions and relationships between closely related patterns is a very challenging and unresolved topic that is currented be addressed by the patterns community.

4 Related Work

[1, 5, 20] 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 the literature. Note that many of the tactical patterns form the basis for implementing the strategic patterns presented in this paper.

The Reactor pattern is related to the Observer pattern [1]. 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 (though there can be multiple sources of events). The Reactor pattern also provides a Facade [1]. 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 mechanism the Reactor uses to dispatch `Event Handlers` is similar to the Factory Callback pattern [21]. The intent of both patterns is to decoupling event reception from event processing. The primary different is the *purpose* of the pattern – 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 [1]. 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 [5]. 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 Strategy and Factory Method patterns [1]. The Acceptor pattern is a connection factory that embodies the strategy for create service handlers, accepting connections into service handlers, and activating service handles to process data exchanged across communication channels. The `Acceptor` 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 variant of the Mediator pattern [1], which decouples cooperating components of a software system and allows them to interact without having direct dependencies among each other. The Router 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 Router pattern allows the use of different concurrency strategies for input and output channels.

5 Concluding Remarks

This paper illustrates how a family of patterns have been applied to facilitate widespread reuse of design expertise and software components in production communication `Gateways`. These patterns 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 family of design patterns and the 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 and electronic medical imaging projects [12, 3] to academic research projects [10]. In general, patterns aid the development of 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.

[6] An ACE-based example of single-threaded and multi-threaded `Gateways` that illustrates all the patterns in this paper is freely available via the WWW at `http://www.cs.wustl.edu/~schmidt`.

Our experience applying a design pattern-based reuse strategy has been quite positive [2]. For instance, we've significantly reduced the software maintenance and training effort for the production communication systems by documenting the intent, structure, and behavior of ACE components in terms of the patterns they reify. Focusing on patterns has also enabled us to reuse software architecture even when reuse of algorithms, implementations, interfaces, or detailed designs was not feasible due to differences in OS platforms [12]. An in-depth discussion of our experiences and lessons learned using patterns appeared in [2].

Acknowledgements

I would like to thank Steve Berczuk, Chris Cleeland, Tim Harrison, Hans Rohnert, and the anonymous referees for contributing valuable suggestions that helped improve the quality of this paper.

References

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

[4] 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.

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

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

[7] 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.

[8] D. C. Schmidt, "Acceptor and Connector: Design Patterns for Initializing Communication Services," in *The 1^{st} European Pattern Languages of Programming Conference (Washington University technical report #WUCS-97-07)*, July 1997.

[9] 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.

[10] 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.

[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] 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.

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

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

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

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

[17] D. C. Schmidt and C. D. Cranor, "Half-Sync/Half-Async: an Architectural Pattern for Efficient and Well-structured Concurrent I/O," in *Pattern Languages of Program Design* (J. O. Coplien, J. Vlissides, and N. Kerth, eds.), Reading, MA: Addison-Wesley, 1996.

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

[19] B. Liskov and L. Shrira, "Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems," in *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation*, pp. 260–267, June 1988.

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

[21] 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.