Applying Design Patterns to Flexibly Configure Network Services in Distributed Systems

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

This paper appeared as a chapter in the book *Design Patterns in Communications*, (Linda Rising, ed.), Cambridge University Press, 2000. An earlier version of this paper appeared in the International Conference on Configurable Distributed Systems, Annapolis, Maryland, May 6–8, 1996.

Abstract

This paper describes how design patterns help to enhance the flexibility and extensibility of communication software by permitting network services to evolve independently of the strategies used to passively initialize the services. The paper makes three contributions to the study and development of configurable distributed applications. First, it identifies five orthogonal dimensions of passive service initialization: service advertisement, endpoint listening, service handler creation, passive connection establishment, and service handler activation. Second, the paper illustrates how design patterns have been used to build a communication software framework that supports flexible configuration of different strategies for each of these five dimensions. Third, the paper demonstrates how design patterns and frameworks are being used successfully to develop highly configurable production distributed systems.

1 Introduction

Despite dramatic increases in network and host performance, developing extensible communication software for distributed systems remains hard. *Design patterns* [1] are a promising technique for capturing and articulating proven techniques for developing extensible distributed software. A design pattern captures the static and dynamic structures and collaborations of components in a software architecture. It also aids the development of extensible components and frameworks by expressing the structure and collaboration of participants in a software architecture at a level higher than (1) source code or (2) object-oriented design models that focus on individual objects and classes.

This paper examines design patterns that form the basis for flexibly configuring network services in applications built by the author and his colleagues for a number of production distributed systems. Due to stringent requirements for reliability and performance, these projects provided an excellent testbed for capturing and articulating the key structure, participants, and consequences of design patterns for building extensible distributed systems.

The primary focus of this paper is the `Acceptor` component in the *Acceptor-Connector* pattern [2]. This design pattern decouples connection establishment and service initialization from service processing in a networked system. The `Acceptor` component is a role in this pattern that enables the tasks performed by network services to evolve independently of the strategies used to initialize the services *passively*.

When instantiated and used in conjunction with other patterns, such as Reactor [2] and Strategy [1], the Acceptor-Connector pattern provides a reusable component in the ACE framework [3]. ACE provides a rich set of reusable object-oriented components that perform common communication softwarekg tasks. These tasks include event demultiplexing, event handler dispatching, connection establishment, routing, dynamic configuration of application services, and concurrency control.

This paper is organized as follows: Section 2 motivates the Acceptor-Connector pattern by illustrating how it has been applied in production application-level `Gateways`; Section 3 outlines the `Acceptor` component of the Acceptor-Connector pattern; Section 4 illustrates how to implement `Acceptor`'s flexibly and efficiently by applying the Wrapper Facade [2], Strategy, Bridge, Factory Method, and Abstract Factory design patterns [1]; Section 5 outlines how `Acceptors` have been used to implement application-level `Gateways`; Section 6 discusses related patterns; and Section 7 presents concluding remarks.

*This research is supported in part by a grant from Siemens MED.

2 Background and Motivation

2.1 Separating Connection establishment and service initialization

Many network services, such as remote login, file transfer, and WWW HTML document transfer, use connection-oriented protocols, such as TCP, to deliver data reliably between two or more communication endpoints. Establishing connections between endpoints involves the following two roles:

1. The *passive* role, which initializes an endpoint of communication at a particular address and waits passively for the other endpoint(s) to connect with it.

2. The *active* role, which actively initiates a connection to one or more endpoints that are playing the passive role.

The intent of the Acceptor-Connector pattern described in this paper is to decouple passive initialization of a service from the tasks performed after the service is initialized. This pattern was discovered by generalizing from extensive experience building reusable communication frameworks for a range of distributed systems [3]. In all these systems, the tasks performed by a service are independent of the following:

- **Which endpoint initiated the connection:** Connection establishment is inherently asymmetrical since the passive endpoint *waits* whereas the active endpoint *initiates* the connection. After the connection is established, however, data may be transferred between endpoints in a manner that obeys a service's communication protocol, which can be structured as peer-to-peer, request-response, oneway streaming, etc.

- **The network programming interfaces and underlying protocols used to establish the connection:** Different network programming interfaces, such as sockets or TLI, provide different routines to establish connections using various underlying transport protocols. After a connection is established, however, data may be transferred between endpoints using standard `read`/`write` system calls that communicate between separate endpoints in a distributed application.

- **The strategies used to initialize the service:** The processing tasks performed by a service are typically independent of the initialization strategies used to (1) advertise the service, (2) listen for connection requests from peers, (3) create a service handler to process those requests, (4) establish the connection with the peers, and (5) execute the service handler in one or more threads or processes. Explicitly decoupling these initialization strategies from the service behavior itself enhances the extensibility, reusability, and portability of the service.

2.2 Motivating Example

Figure 1 illustrates how the Acceptor-Connector pattern has been used to implement multi-service, application-level `Gateways`, which is described further in [4]. A `Gateway`

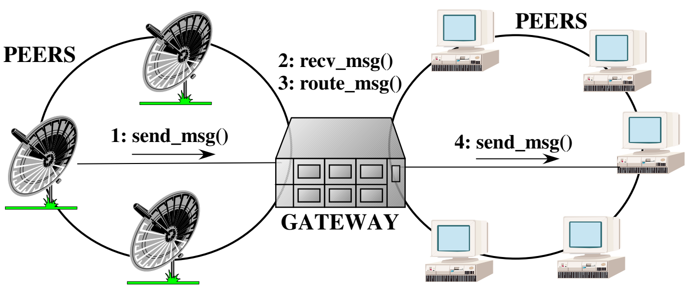

Figure 1: A Connection-oriented, Multi-service Application-level Gateway

is a mediator [1] that routes data between services running on `Peers` located throughout a wide area and local area network. From the `Gateway`'s perspective, `Peer` services differ solely by their message framing formats and payload types. Several types of data, such as status information, bulk data, and commands, are exchanged by services running on the `Gateway` and the `Peers`. `Peers` are located throughout local area networks (LANs) and wide-area networks (WANs) and are used to monitor and control network resources, such as satellites, call centers, or remote branch offices.

The `Gateway` uses a connection-oriented interprocess communication (IPC) mechanism, such as TCP, to transmit data between its connected `Peers`. Connection-oriented protocols simplify application error handling and can enhance performance over long-delay WANs. Each communication service in the `Peers` sends and receives status information, bulk data, and commands to and from the `Gateway` using separate TCP connections. Each connection is bound to a unique address, *e.g.,* an IP address and port number. For instance, bulk data sent from a ground station `Peer` through the `Gateway` is connected to a different port than status information sent by a tracking station peer through the `Gateway` to a ground station `Peer`. Separating connections in this manner allows more flexible routing strategies and more robust error handling if connections fail or become flow controlled.

One way to design the `Peers` and `Gateway` is to tightly couple the connection roles with the network services. For instance, the `Gateway` could be hard-coded to play the active connection role and initiate connections for all its services. To accomplish this, it could iterate through a list of `Peers` and synchronously connect with each of them. Likewise, `Peers` could be hard-coded to play the passive role and accept the connections and initialize their services. Moreover, the active and passive connection code for the `Gateway` and `Peers`, respectively, could be implemented with conventional network programming interfaces like sockets or TLI. In this case, a `Peer` could call `socket`, `bind`, `listen`, and `accept` to initialize a passive-mode listener socket and the `Gateway` could call `socket` and `connect` to actively initiate a data-mode connection socket. After the connections were established, the `Gateway` could route data for each type of service it provided.

However, the tightly coupled design outlined above has the following drawbacks:

• **Limited extensibility and reuse of the Gateway and Peer software:** For example, the mechanisms used to establish connections and initialize services are independent of the type of routing service, *e.g.,* status information, bulk data, or commands, performed by the `Gateway`. In general, these services tend to change more frequently than the connection and initialization mechanisms.

• **Inflexible connection roles:** There are circumstances where the `Gateway` must play the *passive* connection role and the `Peers` play the active role. Therefore, tightly coupling the software that implements connection establishment with the software that implements the service makes it hard to (1) reuse existing services, (2) extend the `Gateway` by adding new routing services and enhancing existing services, and (3) reconfigure the connection roles played by `Peers` and the `Gateway`.

• **Non-portable and error-prone interfaces:** Using low-level network programming, such as sockets or TLI, is non-portable and error-prone. These low-level interfaces do not provide adequate type-checking since they utilize low-level I/O handles. It is easy to accidentally misuse these interfaces in ways that cannot be detected until run-time.

Therefore, a more flexible and efficient way to design the `Peers` and `Gateway` is to use the *Acceptor* pattern.

3 The Acceptor-Connector Pattern

This section presents a brief overview of the Acceptor-Connector pattern. A comprehensive discussion is available in [2].

Intent: The intent of the Acceptor-Connector pattern is to decouple connection establishment and service initialization from service processing in a networked system.

Forces: The Acceptor-Connector pattern resolves the following forces for distributed applications that use connection-oriented communication protocols:

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

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

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

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

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

Structure and participants: The structure of the key participants in the Acceptor-Connector pattern is illustrated in Figure 2. The `Acceptor` and `Connector` components are factories that assemble the resources necessary to connect and activate `Svc_Handlers`. `Svc_Handlers` are components that exchange messages with connected `Peers`.

The participants in the Connection Layer of the Acceptor-Connector pattern can leverage off the Reactor pattern. For instance, the `Connector`'s asynchronous initialization strategy establishes a connection after a `reactor` notifies it that a previously initiated connection request to a `Peer` has completed. Using the Reactor pattern enables multiple `Svc_Handlers`

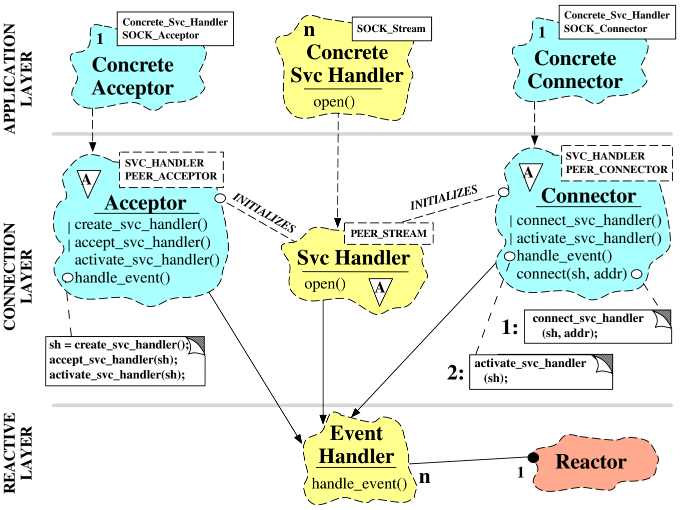

Figure 2: Structure of Participants in the Acceptor-Connector Pattern

to be initialized asynchronously within a single thread of control.

To increase flexibility, `Acceptor` and `Connector` components can be parameterized by a particular type of IPC mechanism and `SVC_HANDLER`. The IPC mechanism supplies the underlying transport mechanism, such as C++ wrapper facades [2] for sockets or TLI, used to establish a connection. The `SVC_HANDLER` specifies an abstract interface for defining a service that communicates with a connected `Peer`. A `Svc_Handler` can be parameterized by a `PEER_STREAM` endpoint. The `Acceptor` and `Connector` components associate this endpoint to its `Peer` when a connection is established.

By inheriting from `Event_Handler`, a `Svc_Handler` can register with a `Reactor` and use the Reactor pattern to handle its I/O events within the same thread of control as the `Acceptor` or `Connector`. Conversely, a `Svc_Handler` can use the Active Object pattern and handle its I/O events in a separate thread. The tradeoffs between these two patterns is described in [4].

Figure 2 illustrates how parameterized types can be used to decouple the Acceptor-Connector pattern's connection establishment strategy from the type of service and the type of connection mechanism. Application developers supply template arguments for these types to produce Application Layer `Acceptor` or `Connectors`. This design enables the wholesale replacement of the `SVC_HANDLER` and IPC mechanism, without affecting the Acceptor-Connector pattern's service initialization strategy.

Note that a similar degree of decoupling could be achieved via inheritance and dynamic binding by using the Abstract Factory or Factory Method patterns described in [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.

Dynamics: Figure 3 illustrates the dynamics among participants for the `Acceptor` component of the pattern. These

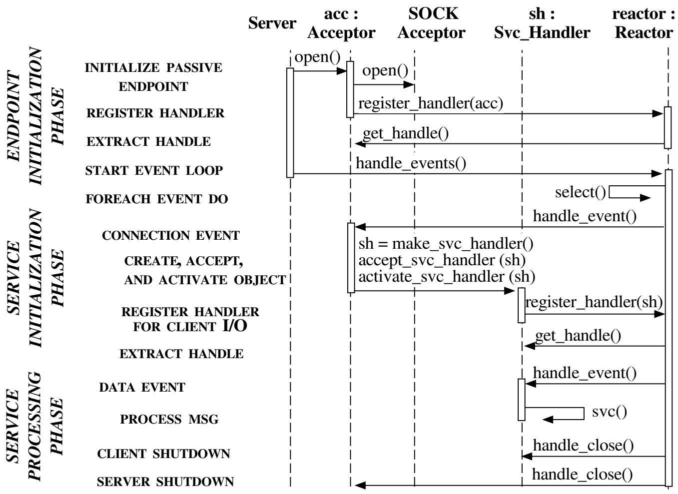

Figure 3: Dynamics for the `Acceptor` Component

dynamics are divided into the following three phases:

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

2. *Service activation phase.* Since an `Acceptor` inherits from an `Event Handler` the `Reactor` can dispatch the `Acceptor`'s `handle_event` method when connection request events arrive. This method performs the `Acceptor`'s `Svc_Handler` initialization strategy, which (1) assembles the resources necessary to create a new `Concrete_Svc_Handler` object, (2) accepts the connection into this object, and (3) activates the `Svc_Handler` by calling its `open` hook method.

3. *Service processing phase.* After the `Svc_Handler` is activated, it processes incoming event messages arriving on the `PEER_STREAM`. A `Svc_Handler` can process incoming event messages in accordance with patterns, such as the Reactor or the Active Object [2].

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

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

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

3. *Service processing phase*, which performs the application-specific service processing using the data exchanged between the `Svc_Handler` and its connected `Peer`.

4 Applying Design Patterns to Develop Extensible Acceptors

This section describes how to implement a highly configurable instance of the `Acceptor` component from the Acceptor-Connector pattern by applying other design patterns, in particular Wrapper Facade [2], Strategy, Bridge, Factory Method, and Abstract Factory [1]. These patterns enable an `Acceptor` to flexibly configure alternative strategies for *service advertisement*, *endpoint listening*, *service handler creation*, *service connection acceptance*, and *service activation*. In this section, we focus on the `Svc_Handler` and the `Acceptor` components shown in Figure 2. The `Connector` component can be implemented in similarly.

4.1 The Svc_Handler Class

This abstract C++ class provides a generic interface for processing services. Applications customize this class to perform a particular type of service. The C++ interface for the `Svc_Handler` is shown below:

```
template <class PEER_STREAM>
         // Type of IPC mechanism.
class Svc_Handler {
public:
  // Pure virtual method (defined by subclass).
  virtual int open (void) = 0;

protected:
  // Instance of IPC mechanism.
  PEER_STREAM stream_;
};
```

Each `Svc_Handler` contains a communication endpoint, called `peer_stream_`, of parameterized type `PEER_STREAM`. This endpoint is used to exchange data between the `Svc_Handler` and its connected peer. After a connection is successfully accepted, an `Acceptor` activates a `Svc_Handler` by calling its `open` method. This pure virtual method must be overridden by a concrete `Svc_Handler` subclass and performs service-specific initializations.

4.2 The Acceptor Class

This abstract C++ class implements the generic strategy for passively initializing network services, which are implemented as concrete `Svc_Handlers`. An `Acceptor` instance coordinates the following five orthogonal dimensions of passive service initialization:

1. *Service advertisement*, which initializes the `peer_acceptor_` endpoint and announces the availability of the service to interested peers.

2. *Endpoint listening*, which waits passively for peers to actively initiate connections on the `peer_acceptor_` endpoint.

3. *Service handler creation*, which creates and initializes a concrete `Svc_Handler` that can communicate with the new peer.

4. *Passive connection establishment*, which uses the `peer_acceptor_` endpoint to accept a connection initiated actively by a peer.

5. *Service handler concurrency activation*, which determines the type of concurrency mechanism used to process data exchanged with peers.

The `Acceptor`'s `open` method is responsible for handling the first two dimensions. The `Acceptor`'s `accept` method is responsible for handling the remaining three dimensions.

The following interface illustrates the methods and data members in the `Acceptor` class:

```
template <class SVC_HANDLER,
          // Type of service handler.
          class PEER_ACCEPTOR>
          // Type of passive connection mechanism.
class Acceptor {
public:
  // Defines the initialization strategies.
  typedef Strategy_Factory<SVC_HANDLER,
                           PEER_ACCEPTOR>
          STRATEGY_FACTORY;

  // Initialize listener endpoint at <addr>
  // according to specified <init_strategies>.
  virtual void open
    (const PEER_ACCEPTOR::PEER_ADDR &addr,
     STRATEGY_FACTORY *init_strategies);

  // Embodies the strategies for creating,
  // connecting, and activating <SVC_HANDLER>'s.
  virtual void accept (void);

protected:
  // Defines strategy to advertise endpoint.
  virtual void advertise_svc
    (const PEER_ACCEPTOR::PEER_ADDR &);
```

```
  // Defines the strategy to listen for active
  // connections from peers.
  virtual void make_listener (PEER_ACCEPTOR *);

  // Defines <SVC_HANDLER> creation strategy.
  virtual SVC_HANDLER *make_svc_handler (void);

  // Defines <SVC_HANDLER> connection strategy.
  virtual void accept_svc_handler (SVC_HANDLER *);

  // Defines <SVC_HANDLER> concurrency strategy.
  virtual int activate_svc_handler (SVC_HANDLER *);

private:
  // Pointers to objects that implement the
  // <Acceptor>'s initialization Strategies.
  Advertise_Strategy<PEER_ACCEPTOR::PEER_ADDR>
    *listen_strategy_;
  Listener_Strategy<PEER_ACCEPTOR>
    *listen_strategy_;
  Creation_Strategy<SVC_HANDLER>
    *create_strategy_;
  Accept_Strategy<SVC_HANDLER, PEER_ACCEPTOR>
    *accept_strategy_;
  Concurrency_Strategy<SVC_HANDLER>
    *concurrency_strategy_;
};
```

The `Acceptor` is a C++ template that is parameterized by a particular type of `PEER_ACCEPTOR` and `SVC_HANDLER`. The `PEER_ACCEPTOR` is the type of transport mechanism used by the `Acceptor` to passively establish the connection. The `SVC_HANDLER` is the type of service that processes the data exchanged with its connected peer. Parameterized types are used to efficiently decouple the service initialization strategies from the type of `Svc_Handler`, network programming interface, and transport layer connection protocol. This design improves the extensibility of the `Acceptor` and `Svc_Handler` components by allowing the wholesale replacement of various strategies.

Figure 4 visually depicts the relationship between the classes that comprise the `Acceptor` implementation. The five strategies supported by the `Acceptor` to passively initialize `Svc_Handlers` are illustrated and described below.

1. Service advertisement strategies: The `Acceptor` uses its service advertisement strategy to initialize the `PEER_ACCEPTOR` endpoint and to announce the availability of the service to interested peers. Figure 5 illustrates the common strategies configured into the `Acceptor` to advertise services:

- *Well-known addresses*, such as Internet port numbers and host names;
- *Endpoint portmappers*, such as those used by Sun RPC and DCE;
- *X.500 directory service*, which is a ISO OSI standard for mapping names to values in a distributed system.

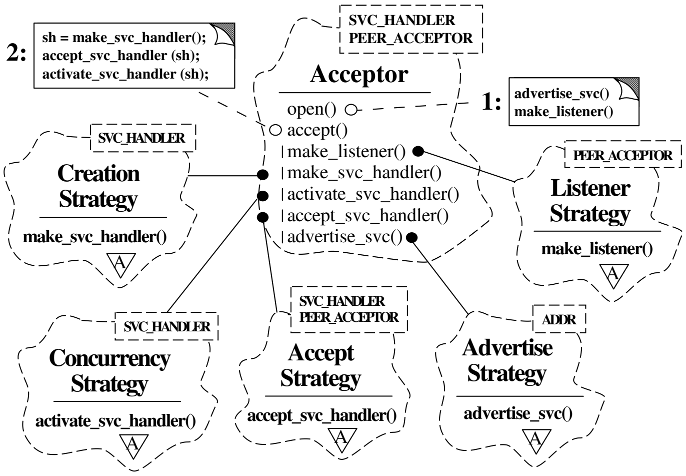

Figure 4: Class Structure of the `Acceptor` Class Implementation

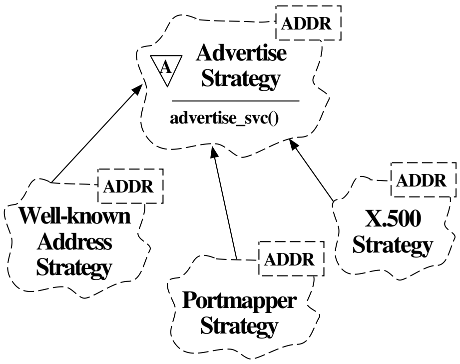

Figure 5: Alternative Service Advertising Strategies

2. Endpoint listener strategies: The `Acceptor` uses its endpoint listening strategy to wait passively for peers to actively initiate a connection to the `PEER_ACCEPTOR` endpoint. Figure 6 illustrates the following common strategies config-

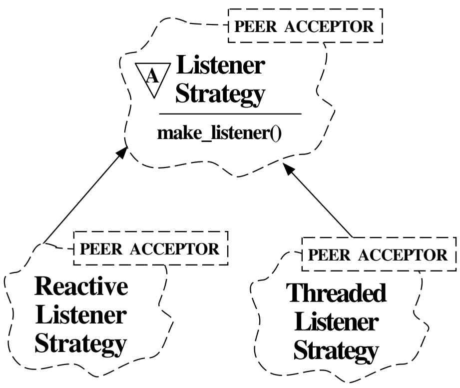

Figure 6: Alternative Endpoint Listener Strategies

ured into the `Acceptor` to wait for connections:

- *Reactive listeners*, which use an event-demultiplexer, such as a `Reactor` [2], to listen passively on a set of endpoints in a single thread of control;
- *Threaded listeners*, which use a separate thread of control for each listener.

3. Service handler creation strategies: The `Acceptor` uses its creation strategy to initialize a `Svc_Handler` that will communicate with the new peer. Figure 7 illustrates the following common strategies configured into the `Acceptor` to create `Svc_Handlers`:

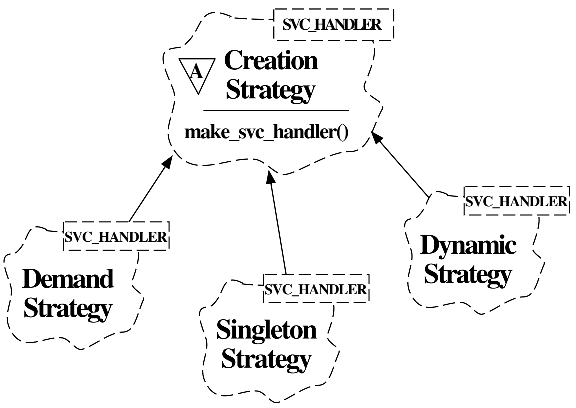

Figure 7: Alternative `Svc_Handler` Creation Strategies

- *Demand creation*, which allocates a new `Svc_Handler` for every new connection;
- *Singleton creation*, which only creates a single `Svc_Handler` that is recycled for every connection;
- *Dynamic creation*, which does not store the `Svc_Handler` object in the application process until it is required, at which point the object is dynamically linked into the process from a shared library.

4. Passive connection establishment strategies: The `Acceptor` uses its passive connection establishment strategy to accept a new connection initiated actively by a peer. Figure 8 illustrates the following common strategies configured into the `Acceptor` to accept connections from peers:

- *Connection-oriented (CONS) establishment*, which uses connection-oriented protocols, such as TCP, SPX, or TP4;
- *Connectionless (CLNS) establishment*, which uses the Adapter pattern [1] to utilize a uniform interface for connectionless protocols.

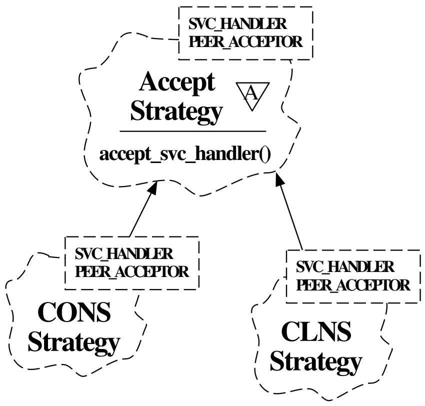

Figure 8: Alternative `Svc_Handler` Connection Acceptance Strategies

5. Service handler concurrency activation strategies: The `Acceptor` uses its activation strategy to determine the type of concurrency mechanism a `Svc_Handler` will use to process data exchanged with its peer. Figure 9 illustrates the following common strategies configured into the `Acceptor` to activate `Svc_Handlers`:

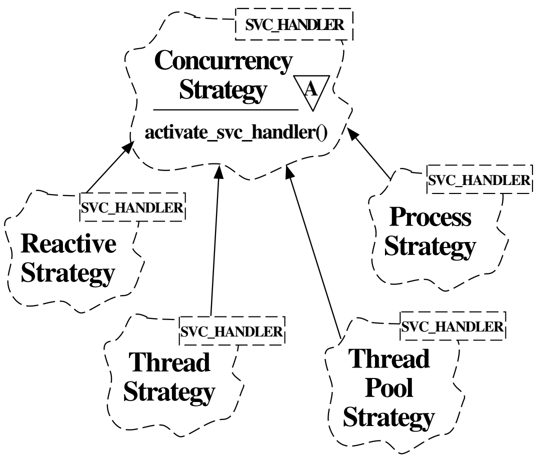

Figure 9: Alternative `Svc_Handler` Concurrency Activation Strategies

- *Reactive activation*, where all `Svc_Handlers` execute within a single thread of control by using the Reactor pattern [2];
- *Thread activation*, where each `Svc_Handler` executes within its own separate thread;
- *Thread pool activation*, where each `Svc_Handler` executes within a pool of threads to increase performance on multi-processors;
- *Process activation*, where each `Svc_Handler` executes within a separate process.

The next section illustrates how different Acceptors can be configured flexibly to support alternative strategies without requiring changes to its external interface design or internal implementation.

4.3 Using Design Patterns to Implement an Extensible Acceptor

The ACE implementation of the Acceptor-Connector pattern applies the Wrapper Facade [2], Factory Method, Strategy, Bridge, and Abstract Factory patterns described in [1]. These patterns facilitate the flexible and extensible configuration and use of the initialization strategies discussed above. Below, each pattern used in the ACE Acceptor is described, the design forces it resolves are outlined, and an example of how the pattern is used to implement the Acceptor is presented.

Using the Wrapper Facades Pattern: The Wrapper Facade [2] pattern encapsulates the functions and data provided by existing non-OO APIs within more concise, robust, portable, maintainable, and cohesive OO class interfaces. The ACE Acceptor uses the Wrapper Facade pattern to provide a uniform interface that encapsulates differences between non-uniform network programming mechanisms, such as sockets, TLI, named pipes, and STREAM pipes.

Figure 10 illustrates how the ACE Acceptor uses the Wrapper Facade patterns to enhance its portability across platforms that contain different network programming interfaces, such as sockets but not TLI, or vice versa. In this example, the PEER_STREAM template argument of the Svc_Handler class can be instantiated with either a SOCK_Stream or a TLI_Stream, depending on whether the platform supports sockets or TLI. The Wrapper Facade pattern ensures that these two classes can be used identically by different instantiations of the Svc_Handler class.

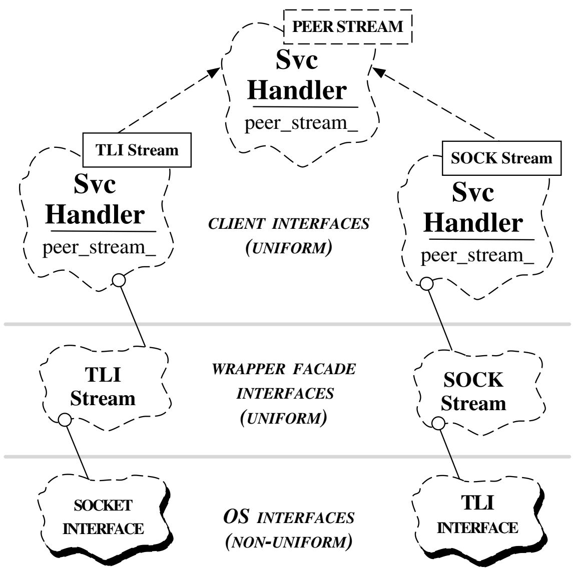

Figure 10: Using the Wrapper Facade Pattern

Using the Strategy Pattern: The Strategy pattern [1] defines a family of algorithms, encapsulates each one as an object, and makes them interchangeable. The ACE Acceptor uses this pattern to determine the passive initialization strategies used to create, accept, and execute a Svc_Handler. By using the Strategy pattern, an application can configure different initialization strategies *without* modifying the following algorithm used by accept, as follows:

```
template <class SVC_HANDLER, class PEER_ACCEPTOR> void
Acceptor<SVC_HANDLER, PEER_ACCEPTOR>::accept (void)
{
  // Create a new <SVC_HANDLER>.
  SVC_HANDLER *svc_handler =
    make_svc_handler ();

  // Accept connection from the peer.
  accept_svc_handler (svc_handler);

  // Activate <SVC_HANDLER>.
  activate_svc_handler (svc_handler);
}
```

Figure 11 illustrates how the Strategy pattern is used to implement the Acceptor's concurrency strategy. When the Acceptor is initialized, its Strategy_Factory configures the designated concurrency strategy. As shown in Figure 9, there are a number of alternative strategies. The particularly strategy illustrated in Figure 11 activates each Svc_Handler to run in a separate thread of control. Since all concurrency algorithms are encapsulated in a uniform interface, however, it is easy to replace this strategy with an alternative one, such as running the Svc_Handler in a separate process.

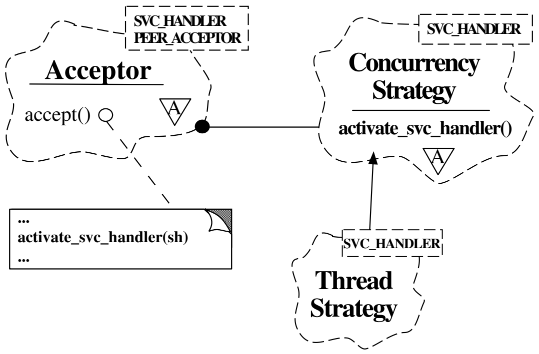

Figure 11: Using the Strategy Pattern

Using the Bridge Pattern: The Bridge pattern [1] decouples an abstraction from its implementation so that the two can vary independently. The ACE Acceptor uses this pattern to provide a stable, uniform interface that is both open (*i.e.,* extensible) and closed (*i.e.,* does not require direct code changes).

Figure 12 illustrates how the Bridge pattern is used to implement the `Acceptor`'s connection acceptance strategy

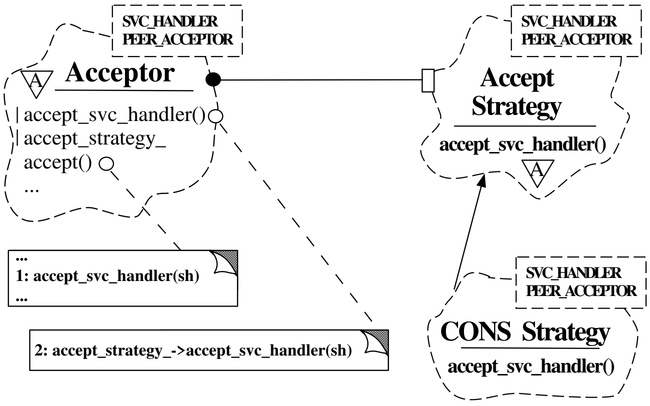

Figure 12: Using the Bridge Pattern

(the Bridge pattern is used for all the other `Acceptor` strategies, as well). When a connection is established with a peer, the `Acceptor`'s `accept` method invokes the `accept_svc_handler` method. Instead of performing the passive connection acceptance strategy directly, however, this method forwards the method to the appropriate subclass of `Accept_Strategy`. In the example shown in Figure 12, this subclass establishes the connection using a connection-oriented protocol. Since the Bridge pattern is used, however, an application can change the `Acceptor`'s connection acceptance strategy to an alternative strategy. For example, it can change to the connectionless version shown in Figure 8 without requiring any changes to the code in `accept`.

Another advantage of using the Bridge pattern is that a subclass of the `Acceptor` can override its `make_*` methods to avoid the additional overhead of indirecting through strategy objects on every call. In this case, the `accept` method uses the Template Method pattern [1]. In the Template Method version of `accept` the steps in the `Acceptor`'s passive initialization algorithm are fixed, but can be overridden by derived classes.

Using the Factory Method Pattern: The Factory Method pattern [1] defines a stable interface for initializing a component, but allows subclasses to specify the details of the initialization. The ACE `Acceptor` uses this pattern to allow each initialization strategy used by the `Acceptor` to be extended without modifying the `Acceptor` or `Svc_Handler` implementations.

Figure 13 illustrates how the Factory Method pattern is used to transparently extend the `Acceptor`'s creation strategy. The `Creation_Strategy` base class contains a

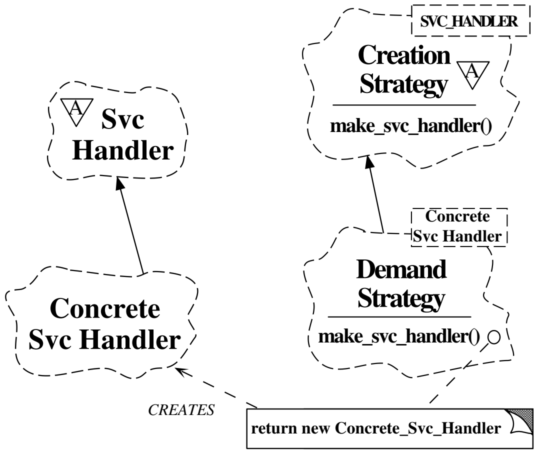

Figure 13: Using the Factory Method Pattern

factory method called `make_svc_handler`. This method is invoked by the `make_svc_handler` Bridge method in the `Acceptor` to create the appropriate type of concrete `Svc_Handler`, as follows:

```
template <class SVC_HANDLER, class PEER_ACCEPTOR> void
Acceptor<SVC_HANDLER, PEER_ACCEPTOR>::accept (void)
{
  creation_strategy_->make_svc_handler ();
}
```

An implementation of a creation strategy based on the *demand* strategy could be implemented as follows:

```
template <class SVC_HANDLER> SVC_HANDLER *
Demand_Strategy<SVC_HANDLER>::make_svc_handler (void) {
  // Implement the ''demand'' creation
  // strategy by allocating a new <SVC_HANDLER>.
  return new SVC_HANDLER;
}
```

Note that it is the responsibility of the `Acceptor`'s `Strategy_Factory` to determine the type of subclass associated with the `creation_strategy_`.

Using the Abstract Factory: The Abstract Factory pattern [1] provides a single interface that creates families of related objects without requiring the specification of their concrete classes. The `Acceptor` uses this pattern to simplify its interface by localizing all five of its initialization strategies into a single class. The Abstract Factory pattern also ensures that all selected strategies can work together correctly.

Figure 14 illustrates how the Abstract Factory pattern is used to implement the `Status_Acceptor` taken from the `Gateway` example describe in Section 5. This example instantiates the following `Strategy_Factory` template:

```
template <class SVC_HANDLER,
          // Type of service handler.
          class PEER_ACCEPTOR>
          // Type of passive connection.
```

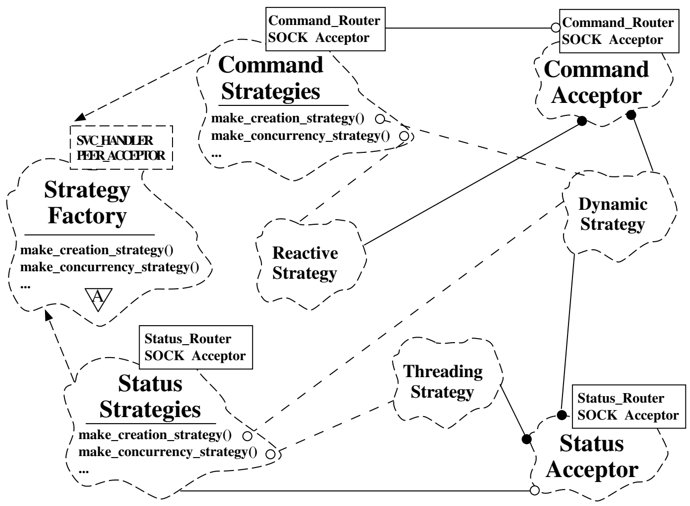

Figure 14: Using the Abstract Factory Pattern

```
class Strategy_Factory {
public:
  Strategy_Factory
    (Advertise_Strategy<PEER_ACCEPTOR::PEER_ADDR> *,
     Listener_Strategy<PEER_ACCEPTOR> *,
     Creation_Strategy<SVC_HANDLER> *,
     Accept_Strategy<SVC_HANDLER, PEER_ACCEPTOR> *,
     Concurrency_Strategy<SVC_HANDLER> *);

  // Factory methods called by Acceptor::open().
  Advertise_Strategy<PEER_ACCEPTOR::PEER_ADDR>
    *make_advertise_strategy (void);
  Listener_Strategy<PEER_ACCEPTOR>
    *make_listener_strategy (void);
  Creation_Strategy<SVC_HANDLER>
    *make_create_strategy (void);
  Accept_Strategy<SVC_HANDLER, PEER_ACCEPTOR>
    *make_accept_strategy (void);
  Concurrency_Strategy<SVC_HANDLER>
    *make_concurrency_strategy (void);

  // ...
```

Figure 14 shows the creation and concurrency strategies–the other strategies are handled similarly. The `Status_Strategies` factory instructs the `Status_Acceptor` to dynamically create each `Status_Router`, which will execute in its own thread of control. This example illustrates the following points:

- The Abstract Factory pattern is often used in conjunction with the Factory Method pattern. For example, the `Strategy_Factory` abstract factory simplifies the interface to the `Acceptor` by consolidating all five initialization strategy factory methods in a single class.

- The Abstract Factory pattern ensures that various the strategies can work together correctly. For instance, the `Strategy_Factory` can be subclassed and its various `make_*` Factory Methods can be overridden to create different types of initialization strategies.

- Subclasses of the `Strategy_Factory` abstract factory can be used to ensure that conflicting initialization strategies are not configured accidentally. For example, the *singleton* creation strategy may conflict with the *thread* concurrency strategy since multiple threads of control will attempt to access a single communication endpoint. A `Strategy_Factory` subclass can be defined to check for these conflicts and report an error at configuration time.

5 Example: Implementing Extensible Application-level Gateways Using the Acceptor

This section illustrates how the application-level `Gateway` described in Section 2 uses the pattern-based `Acceptor` component from Section 4 to simplify the task of passively initializing services whose connections are initiated actively by `Peers`. In this example the `Peers` play the active role in establishing connections with the `Gateway`.

Defining Svc_Handlers for routing peer messages: The three classes shown below, `Status_Router`, `Bulk_Data_Router`, and `Command_Router`, process routing messages received from `Peers`. These classes inherit from `Svc_Handler`, which allows them to be passively initialized by an `Acceptor`, as shown in Figure 15. Each

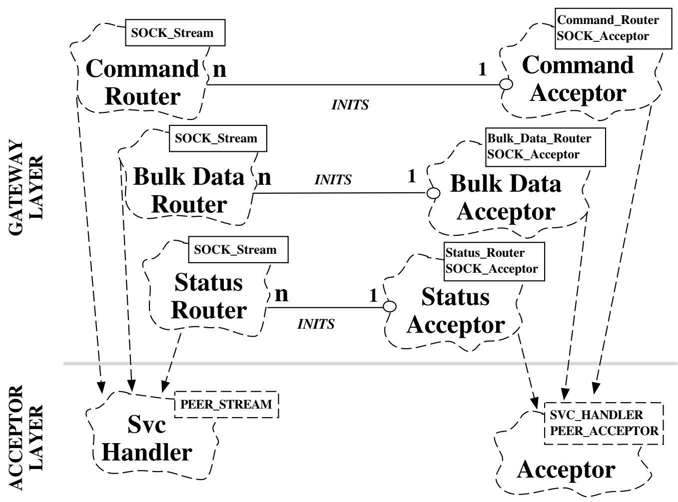

Figure 15: Structure of `Acceptor` Participants in the `Gateway`

class is instantiated with a specific type of C++ IPC wrapper facade that exchanges data with its connected peer. For example, the classes below use a `SOCK_Stream` as the underlying data transport delivery mechanism. `SOCK_Stream` is an ACE C++ wrapper facade that encapsulates the data transfer functions in the socket interface. By virtue of the Strategy pattern, however, it easy to vary the data transfer mechanism by parameterizing the `Svc_Handler` with a different `PEER_STREAM`, such as a `TLI_Stream`.

The `Status_Router` class routes status data sent to and received from `Peers`:[1]

```
class Status_Router :
  public Svc_Handler<SOCK_Stream>
{
public:
  // Performs router initialization.
  virtual int open (void);
  // Receive and route status data from/to peers.
  virtual int handle_event (void);
  // ...
```

The `Bulk_Data_Router` class routes bulk data sent to and received from `Peers`.

```
class Bulk_Data_Router :
  public Svc_Handler<SOCK_Stream>
{
public:
  // Performs router initialization.
  virtual int open (void);
  // Receive and route bulk data from/to peers.
  virtual int handle_event (void);
  // ...
```

The `Command_Router` class routes bulk data sent to and received from `Peers`:

```
class Command_Router :
  public Svc_Handler<SOCK_Stream>
{
public:
  // Performs router initialization.
  virtual int open (void);
  // Receive and route command data from/to peers.
  virtual int handle_event (void);
  //...
```

Defining Acceptor factories to create Svc_Handlers: The three classes shown below are instantiations of the `Acceptor` template:

```
// Typedefs that instantiate <Acceptor>s for
// different types of routers.
typedef Acceptor<Status_Router, SOCK_Acceptor>
        Status_Acceptor;
typedef Acceptor<Bulk_Data_Router, SOCK_Acceptor>
        Bulk_Data_Acceptor
typedef Acceptor<Command_Router, SOCK_Acceptor>
        Command_Acceptor;
```

[1]To save space, these examples have been simplified by omitting most of the detailed protocol logic and error handling code.

These typedefs instantiate the `Acceptor` template with concrete parameterized type arguments for `SVC_HANDLER` and `PEER_ACCEPTOR`. A `SOCK_Acceptor` wrapper facade is used as the underlying `PEER_ACCEPTOR` in order to accept a connection passively. Parameterizing the `Acceptor` with a different `PEER_ACCEPTOR`, such as a `TLI_Acceptor`, is easy since the IPC mechanisms are encapsulated in C++ wrapper facade classes. The three objects shown below are instances of these classes that create and activate `Status_Routers`, `Bulk_Data_Routers`, and `Command_Routers`, respectively:

```
// Accept connection requests from
// Gateway and activate Status_Router.
static Status_Acceptor status_acceptor;

// Accept connection requests from
// Gateway and activate Bulk_Data_Router.
static Bulk_Data_Acceptor bulk_data_acceptor;

// Accept connection requests from
// Gateway and activate Command_Router.
static Command_Acceptor command_acceptor;
```

Defining strategies to initialize Svc_Handlers: The three classes shown below are instantiations of the `Strategy_Factory` described in Section 4.2:

```
// Typedefs that instantiate different types
// of <Strategy_Factory>.
typedef Strategy_Factory<Status_Router,
                         SOCK_Acceptor>
        Status_Strategies;
typedef Strategy_Factory<Bulk_Data_Router,
                         SOCK_Acceptor>
        Bulk_Data_Strategies;
typedef Strategy_Factory<Command_Router,
                         SOCK_Acceptor>
        Command_Strategies;
```

These typedefs instantiate the `Strategy_Factory` template with concrete parameterized type arguments for `SVC_HANDLER` and `PEER_ACCEPTOR`. The three objects shown below instantiate these classes to specify the initialization strategies for `Status_Routers`, `Bulk_Data_Routers`, and `Command_Routers`, respectively:

```
// Creates a multi-threaded <Status_Router>.
Status_Strategies threaded
  (new Well_Known_Addr,
   new Reactive_Listener (Reactor::instance ()),
   new Demand,
   new CONS,
   new Multi_Thread);

// Creates a multi-processed <Bulk_Data_Router>.
Bulk_Data_Strategies process
  (new Well_Known_Addr,
   new Reactive_Listener (Reactor::instance ()),
   new Demand,
   new CONS,
   new Multi_Process);
```

```
// Creates a single-thread reactive <Command_Router>.
Command_Strategies reactive
   (new Well_Known_Addr,
    new Reactive_Listener (Reactor::instance ()),
    new Demand,
    new CONS,
    new Reactive (Reactor::instance ());
```

Each Strategy_Factory configuration shown above uses the *well known address* service advertisement strategy, the *reactive* listener strategy, the *demand* Svc_Handler creation strategy, and the *connection-oriented* acceptance strategy. To illustrate the flexibility of the Acceptor-Connector pattern, however, each Strategy_Factory implements a different concurrency strategy, as follows:

- When the Status_Router is activated by Status_Acceptor it runs in a separate thread.
- When activated by Bulk_Data_Acceptor, the Bulk_Data_Router runs as a separate process.
- When activated by Command_Acceptor, the Command_Router runs in the same thread as with the Reactor singleton [1], which is used to demultiplex connection requests for the three Acceptor factories.

Note how changing the concurrency strategy does not affect the Acceptor class. Thus, the Acceptor's generic strategy for passively initializing services can be reused, while permitting specific details, such as the PEER_ACCEPTOR, SVC_HANDLER, and selected initialization strategies, to change flexibly.

The main() gateway function: The main gateway initializes the Acceptors with their well-known ports and initialization strategies, as follows:

```
// Main program for the Gateway.

int main (void) {
  // Initialize Acceptors with their well-known
  // ports and their initialization strategies.
  status_acceptor.open
   (INET_Addr (STATUS_PORT), &threaded);
  bulk_data_acceptor.open
   (INET_Addr (BULK_DATA_PORT), &process);
  command_acceptor.open
   (INET_Addr (COMMAND_PORT), &reactive);

  // Loop forever handling connection request
  // events and processing data from peers.
  for (;;)
    Reactor::instance ()->handle_events ();
}
```

The listener strategy configured for each Acceptor is reactive, as shown in Section 5. Therefore, the program enters an event loop that uses the Reactor singleton to detect all connection requests from Peers within a single thread of control. When connections arrive, the Reactor singleton dispatches the associated Acceptor, which (1) creates an appropriate type of Svc_Handler on demand to perform the service, (2) accepts the connection into the handler, and (3) activates the handler. The concurrency strategy configured into each Acceptor dictates how every Svc_Handler it creates will processes events.

Figure 16 illustrates the relationship between Acceptor-Connector pattern components in the Gateway after

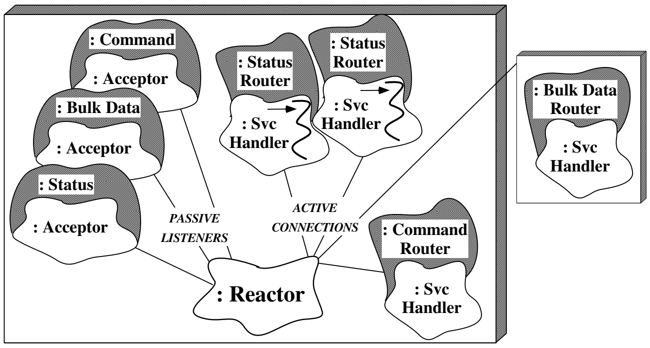

Figure 16: Object Diagram for the Gateway Acceptor-Connector Pattern

four connections have been established. The various *Routers exchange data with their connected Peers using the type of concurrency strategy designated by their Strategy_Factories. Meanwhile, the *Acceptors continue to listen for new connections.

5.1 Known Uses

UNIX network superservers: Superserver implementations such as Inetd [5], Listen [6] and the Service Configurator [7] from the ACE framework use a master acceptor process that listens for connections on a set of communication ports. In Inetd, for example, each port is associated with a service, such as the standard Internet services FTP, TELNET, DAYTIME, and ECHO. The acceptor process decouples the functionality of the Inetd superserver into two separate parts: one for establishing connections and another for receiving and processing requests from peers. When a service request arrives on a port monitored by Inetd, it accepts the request and dispatches an appropriate pre-registered handler to perform the service.

CORBA Object Request Brokers (ORBs): The ORB Core layer in many implementations of CORBA [8] uses the Acceptor-Connector pattern to passively and actively initialize connection handlers when clients request ORB services. For example, [9] describes how the Acceptor-Connector pattern is used to implement the ORB Core portion in The ACE ORB (TAO), which is a high-performance and real-time implementation of CORBA.

Web Browsers: The HTML parsing components in Web browsers such as Netscape and Internet Explorer use the asynchronous version of the connector component to establish connections with servers associated with images embedded in HTML pages. This pattern allows multiple HTTP connections to be initiated asynchronously. This avoids the possibility of the browser's main event loop blocking.

Ericsson EOS Call Center Management System: This system uses the Acceptor-Connector pattern to allow application-level Call Center Manager event servers [10] to establish connections actively with passive supervisors in a networked center management system.

Project Spectrum: The high-speed medical image transfer subsystem of project Spectrum [11] uses the Acceptor-Connector pattern to establish connections passively and initialize application services for storing large medical images. Once connections are established, applications send and receive multi-megabyte medical images to and from the image stores.

ACE: Implementations of the generic `Svc_Handler`, `Connector`, and `Acceptor` components described in the Implementation section are provided as reusable C++ classes in the ACE framework [3]. Java ACE [12] is a version of ACE implemented in Java that provides components corresponding to the participants the Acceptor-Connector pattern.

6 Related Patterns

[1, 13, 2] identify and catalog many architectural and design patterns. This section examines how the patterns described in this paper relate to other patterns in the literature.

The intent of the Acceptor-Connector pattern is similar to the Configuration pattern [14]. The Configuration pattern decouples structural issues related to configuring services in distributed applications from the execution of the services themselves. This pattern has been used in frameworks for configuring distributed systems, such as Regis [15], to support the construction of a distributed system from a set of components. In a similar way, the Acceptor-Connector pattern decouples service initialization from service processing. The primary difference is that the Configuration pattern focuses more on the active composition of a chain of related services, whereas the Acceptor-Connector pattern focuses on the passive initialization of a service handler at a particular endpoint. In addition, the Acceptor-Connector pattern also focuses on decoupling service behavior from the service's concurrency strategies.

The intent of the Acceptor-Connector pattern is similar to that of the Client-Dispatcher-Server pattern [13] in that both are concerned with the separation of active connection establishment from subsequent service processing. The primary difference is that the Acceptor-Connector pattern addresses passive and active connection establishment and initialization of both synchronous and asynchronous connections. In contrast, the Client-Dispatcher-Server pattern focuses on synchronous connection establishment.

The service handlers that are created by acceptors and connectors can be coordinated using the Abstract Session pattern [16], which allows a server object to maintain state for many clients. Likewise, the Half Object plus Protocol pattern [17] can help decompose the responsibilities of an end-to-end service into service handler interfaces and the protocol used to collaborate between them.

The Acceptor-Connector pattern may be viewed as an object creational pattern [1]. A creational pattern assembles the resources necessary to create an object and decouples the creation and initialization of the object from subsequent use of the object. The Acceptor-Connector pattern is a factory that creates, passively connects, and initializes service handlers. Its `accept` method implements the algorithm that listens passively for connection requests, then creates, accepts, and activates a handler when the connection is established. The handler performs a service using data exchanged on the connection. Thus, the subsequent behavior of the service is decoupled from its initialization strategies.

7 Concluding Remarks

This paper describes the Acceptor-Connector pattern and illustrates how its `Acceptor` component has been implemented using other patterns to develop highly flexible communication software. In general, the Acceptor-Connector pattern is applicable whenever connection-oriented applications have the following characteristics:

- The behavior of a distributed service does not depend on the steps required to passively or actively connect and initialize a service.
- Connection requests from different peers may arrive concurrently, but blocking or continuous polling for incoming connections on any individual peer is inefficient.

The Acceptor-Connector pattern provides the following benefits for network applications and services:

It enhances the reusability, portability, and extensibility of connection-oriented software: The Acceptor-Connector pattern decouples mechanisms for connection establishment and service initialization, which are application-independent and thus reusable, from the services themselves, which are application-specific. For example, the application-independent mechanisms in the `Acceptor` are reusable components that know how to establish a connection passively and to create and activate its associated `Svc_Handler`. In contrast, the `Svc_Handler` knows how to perform application-specific service processing.

This separation of concerns decouples connection establishment from service handling, thereby allowing each part to evolve independently. The strategy for establishing connections actively was written once, placed into the ACE framework, and reused via inheritance, object composition, and template instantiation. Thus, the same passive connection establishment code need not be rewritten for each application. In contrast, services may vary according to different application requirements. By parameterizing the `Acceptor` with a `Svc_Handler`, the impact of this variation is localized to a single point in the software.

Improves application robustness: By strongly decoupling the `Acceptor` from the `Svc_Handler` the passive-mode `PEER_ACCEPTOR` cannot accidentally be used to read or write data. This eliminates a class of subtle errors that can arise when programming with weakly typed network programming interfaces such as sockets or TLI.

However, the Acceptor-Connector pattern can also exhibit the following drawbacks:

Additional indirection: The Acceptor-Connector pattern can incur additional indirection compared to using the underlying network programming interfaces directly. However, languages that support parameterized types, such as C++, Ada or Eiffel, can implement these patterns with no significant overhead when compilers inline the method calls used to implement the patterns.

Additional complexity: The Acceptor-Connector pattern may add unnecessary complexity for simple client applications that connect with only one server and perform one service using a single network programming interface. However, the use of generic acceptor and connector wrapper facades may simplify even these use cases by shielding developers from tedious, error-prone and non-portable low-level network programming mechanisms.

Open-source implementations of the Acceptor-Connector and Reactor patterns are available at URL `www.cs.wustl.edu/~schmidt/ACE.html`. This URL contains complete source code, documentation, and example test drivers for the C++ components developed as part of the ACE framework [3] developed at the University of California, Irvine and Washington University, St. Louis.

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

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

[4] D. C. Schmidt, "Applying a Pattern Language to Develop Application-level Gateways," in *Design Patterns in Communications* (L. Rising, ed.), Cambridge University Press, 2000.

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

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

[7] P. Jain and D. C. Schmidt, "Service Configurator: A Pattern for Dynamic Configuration of Services," in *Proceedings of the 3^{rd} Conference on Object-Oriented Technologies and Systems*, USENIX, June 1997.

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

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

[10] D. C. Schmidt and T. Suda, "An Object-Oriented Framework for Dynamically Configuring Extensible Distributed Communication Systems," *IEE/BCS Distributed Systems Engineering Journal (Special Issue on Configurable Distributed Systems)*, vol. 2, pp. 280–293, December 1994.

[11] G. Blaine, M. Boyd, and S. Crider, "Project Spectrum: Scalable Bandwidth for the BJC Health System," *HIMSS, Health Care Communications*, pp. 71–81, 1994.

[12] P. Jain and D. Schmidt, "Experiences Converting a C++ Communication Software Framework to Java," *C++ Report*, vol. 9, January 1997.

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

[14] S. Crane, J. Magee, and N. Pryce, "Design Patterns for Binding in Distributed Systems," in *The OOPSLA '95 Workshop on Design Patterns for Concurrent, Parallel, and Distributed Object-Oriented Systems*, (Austin, TX), ACM, Oct. 1995.

[15] J. Magee, N. Dulay, and J. Kramer, "A Constructive Development Environment for Parallel and Distributed Programs," in *Proceedings of the 2^{nd} International Workshop on Configurable Distributed Systems*, (Pittsburgh, PA), pp. 1–14, IEEE, Mar. 1994.

[16] N. Pryce, "Abstract Session," in *Pattern Languages of Program Design* (B. Foote, N. Harrison, and H. Rohnert, eds.), Reading, MA: Addison-Wesley, 1999.

[17] G. Meszaros, "Half Object plus Protocol," in *Pattern Languages of Program Design* (J. O. Coplien and D. C. Schmidt, eds.), Reading, MA: Addison-Wesley, 1995.