Acceptor

A Design Pattern for Passively Initializing Network Services

Douglas C. Schmidt
schmidt@cs.wustl.edu
Department of Computer Science
Washington University
St. Louis, MO 63130, USA
(314) 935-7538

This paper appeared in the November/December issue of the C++ Report magazine.

1 Introduction

This article is part of a continuing series that describes object-oriented techniques for developing reusable, extensible, and efficient communication software. The current topic examines the *Acceptor* pattern. This design pattern enables the tasks performed by network services to evolve independently of the strategies used to *passively* initialize the services. By decoupling service *initialization* from service *processing*, this pattern enables the creation of reusable, extensible, and efficient network services. When used in conjunction with related patterns like the Reactor [1] and the Connector [2], this pattern enables the creation of highly extensible and efficient communication software frameworks [3].

A companion article [2] examines the Connector pattern, which is the "dual" of the Acceptor pattern. The Connector pattern decouples the active establishment of a connection from the service performed once the connection is established. Although these two patterns address similar forces they are described separately since their structure differs somewhat due to the asymmetry of connection establishment protocols. In addition, the Connector pattern addresses additional forces by using asynchrony to actively establish connections with a large number of peers efficiently.

This article is organized as follows: Section 2 describes the two primary connection roles (active and passive) used to establish connections and explains how these roles can be decoupled from the communication roles performed once connections are established; Section 3 motivates the Acceptor pattern by applying it to a connection-oriented, multi-service, application-level `Gateway`; Section 4 describes the Acceptor pattern in detail and illustrates how to implement it flexibly and efficiently by combining existing design patterns [4] and C++ language features, and Section 5 presents concluding remarks.

2 Background

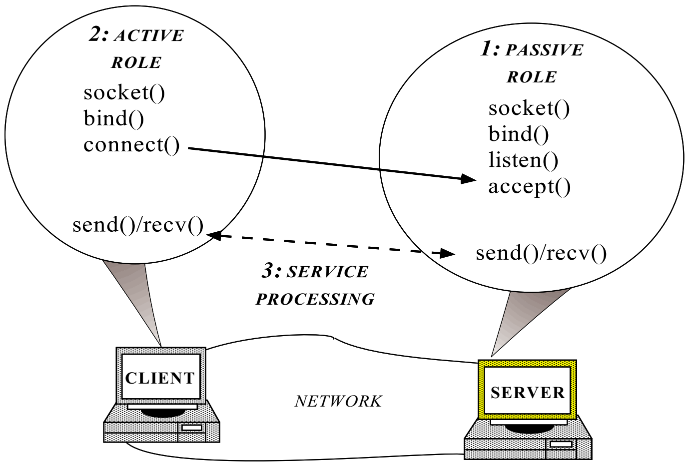

Figure 1: Active and Passive Connection roles

Connection-oriented protocols (such as TCP, SPX, or TP4) reliably deliver data between two or more endpoints of communication. Establishing connections between endpoints involves the following two roles:

- *Passive role* – which initializes an endpoint of communication at a particular address and waits passively for the other endpoint(s) to connect with it;
- *Active role* – which actively initiates a connection to one or more endpoints that are playing the passive role.

Figure 1 illustrates how these connection roles behave and interact when a connection is established between an active client and a passive server using the socket network programming interface [5] and the TCP transport protocol [6]. In this figure the server plays the passive role and the client plays the active role.[1]

One goal of the Acceptor and Connector patterns is to decouple the passive and active connection roles, respectively, from the services performed once a connection is established.

[1] It is important to recognize that traditional distinctions between "client" and "server" refer to *communication* roles, not necessarily to *connection* roles. Although clients often take the active role in establishing connections with a passive server these connection roles can be reversed, as shown in Section 3.

These patterns are motivated by observing that the service processing performed on messages exchanged between connected endpoints is largely independent of the following:

- *Which endpoint initiated the connection* – connection establishment is inherently asymmetrical since the passive endpoint *waits* and the active endpoint *initiates* the connection. Once the connection is established, however, data may be transferred between services at the endpoints in any manner that obeys the application's communication protocol (*e.g.,* peer-to-peer, request-response, oneway streaming, etc.). This is illustrated in Figure 1 by the peer-to-peer `send/recv` communication between client and server once a connection is established.
- *The network programming interfaces and underlying protocols used to establish the connection* – different network programming interfaces (such as sockets [7] or TLI [8]) provide different library calls to establish connections using various underlying transport protocols (such as TCP, TP4, or SPX). Once a connection is established, however, data may be transferred between endpoints using standard `read/write` system calls that obey the protocols used to communicate between separate endpoints in a distributed application.
- *The creation, connection, and concurrency strategies used to initialize and execute the service* – the processing tasks performed by a service typically do not depend on the strategies used to create a service, connect the service to one or more peers, and execute the service in one or more threads or processes. Explicitly decoupling these initialization strategies from the service behavior itself increases the potential for reusing and extending the service in different environments.

3 Motivation

To illustrate the Acceptor and Connector patterns, consider the multi-service, application-level `Gateway` shown in Figure 2. This `Gateway` routes several types of data (such as status information, bulk data, and commands) between services running on `Peers` located throughout a wide area and local area network. The `Gateway` routes several types of data (such as status information, bulk data, and commands) that are exchanged between services running on the `Peers`. These `Peers` are located throughout local area networks (LANs) and wide-area networks (WANs) and are used to monitor and control a satellite constellation.

The `Gateway` is a Mediator [4] that coordinates interactions between its connected `Peers`. From the `Gateway`'s perspective, these `Peer` services differ solely by their message framing formats and payload types. The `Gateway` uses a connection-oriented interprocess communication (IPC) mechanism (such as TCP) to transmit data between its connected `Peers`. Using a connection-oriented protocol simplifies application error handling and enhances performance over long-delay WANs.

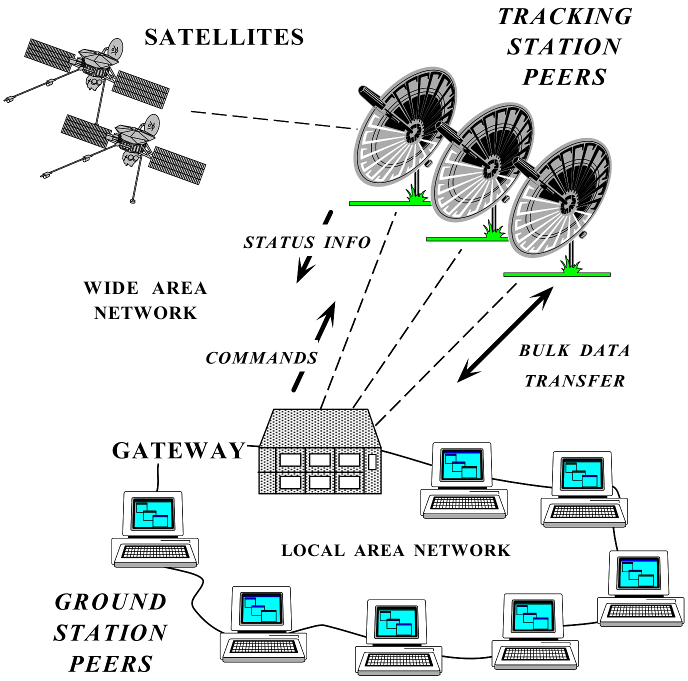

Figure 2: A Connection-oriented, Multi-service Application-level Gateway

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 example, 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 when connections fail.

One way to design the `Peers` and `Gateway` is to designate the connection roles *a priori*. For instance, the `Gateway` could be hard-coded to actively initiate the 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 passively 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 (such as 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. Once the connections were established, the `Gateway` could route data for each type of service it provided.

However, the approach outlined above has several drawbacks:

- *Limited extensibility and reuse* of the `Gateway` and `Peer` software. For example, the type of routing service (*e.g.,* status information, bulk data, or commands) performed by the `Gateway` is independent of the mechanisms used to establish the connection. Moreover, these services tend to change more frequently than the connection mechanisms. Therefore, tightly coupling the software that implements connection establishment with the software that implements the service makes it hard to reuse existing services or to extend the `Gateway` by adding new routing services and enhancing existing services.
- *Error-prone network programming interfaces* – low-level network programming (such as sockets or TLI) do not provide adequate type-checking since they utilize low-level I/O handles [9]. It is surprisingly easy to accidentally misuse these interfaces in ways that cannot be detected until run-time.
- *Lack of scalability* – If there are a large number of `Peers` the synchronous connection establishment strategy of the `Gateway` will not take advantage of the parallelism inherent in the network and `Peers`.

Therefore, a more flexible and efficient way to design the `Peers` and `Gateway` is to use the *Acceptor* and *Connector* patterns. These patterns resolve the following forces for network clients and servers that explicitly use connection-oriented communication protocols:

- *How to enable flexible strategies for executing network services concurrently* – 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.). However, a service can run in a single-thread, in multiple threads, or multiple processes, regardless of how the connection was established.
- *How to reuse existing initialization code for each new service* – The Connector and Acceptor patterns permit key characteristics of services (such as the application-level communication protocol and data format) to evolve independently and transparently from the strategies used to initialize the services. Since service characteristics tend to change more frequently than initialization strategies this separation of concerns helps reduce software coupling and increases code reuse.
- *How to actively establish connections with large number of peers efficiently* – The Connector pattern can employ asynchrony to initiate and complete multiple connections in non-blocking mode. By using asynchrony, the Connector pattern enables applications to actively establish connections with a large number of peers efficiently over long-delay WANs.
- *How to make the connection establishment code portable across platforms that contain different network programming interfaces* – This is important for asynchronous connection establishment, which is hard to program portably and correctly using lower-level network programming interfaces (such as sockets and TLI). Likewise, parameterizing the 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).
- *How 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 accidentally be used incorrectly (*e.g.,* to try to read or write data on a passive-mode listener socket used to accept connections).

Section 4 describes the Acceptor pattern in detail. The Connector pattern is described in [2].

4 The Acceptor Pattern

4.1 Intent

Decouples the passive initialization of a service from the tasks performed once a service is initialized.

4.2 Also Known As

Listener

4.3 Applicability

Use the Acceptor pattern when connection-oriented applications have the following characteristics:

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

4.4 Structure and Participants

The structure of the participants in the Acceptor pattern is illustrated by the Booch class diagram [10] in Figure 3 and described below:[2]

- **Reactor**

[2] In this diagram dashed clouds indicate classes; dashed boxes in the clouds indicate template parameters; and a solid undirected edge with a hollow circle at one end indicates a uses relation between two classes.

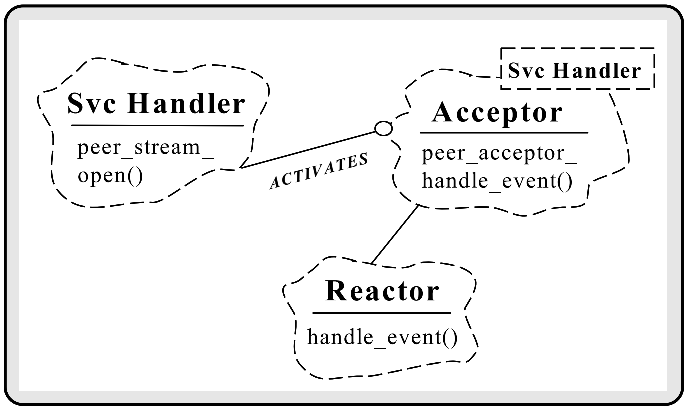

Figure 3: Structure of Participants in the Acceptor Pattern

- Demultiplexes connection requests received on one or more communication endpoints to the appropriate `Accector`. The `Reactor` allows multiple `Acceptors` to listen for connections from peers within a single thread of control.

- **Acceptor**
  - Passively accepts connections from peers using the `peer_acceptor_` endpoint, then creates and activates a `Svc Handler`. The `Acceptor`'s `handle_event` method implements the strategy for initializing a `Svc Handler` by passively connecting it with a peer. The `Reactor` calls back this method automatically when a connection arrives for the `Acceptor`.

- **Svc Handler**
  - Defines a generic interface for a service. The `Svc Handler` contains a communication endpoint (`peer_stream_`) that encapsulates an I/O handle (also known as an "I/O descriptor"). This endpoint is used to exchange data between the `Svc Handler` and its connected peer. The `Acceptor` activates the `Svc Handler`'s `peer_stream_` endpoint by calling its `open` method when a connection completes successfully.

4.5 Collaborations

Figure 4 illustrates the collaboration between participants in the Acceptor pattern. These collaborations are divided into three phases:

1. *Endpoint initialization phase* – which creates a passive-mode endpoint 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.
2. *Service initialization phase* – which activates a `Svc Handler`. When a connection arrives the `Reactor` calls back to the `Acceptor`'s `handle_event` method. This method performs the strategy for initializing a `Svc Handler`. This involves assembling 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. The `open` method of the `Svc Handler` then performs service-specific initialization.
3. *Service processing phase* – Once the connection has been established passively and the service has been initialized, the application enters into a *service processing phase*. This phase performs application-specific tasks that process the data exchanged between the `Svc Handler` and its connected peer(s).

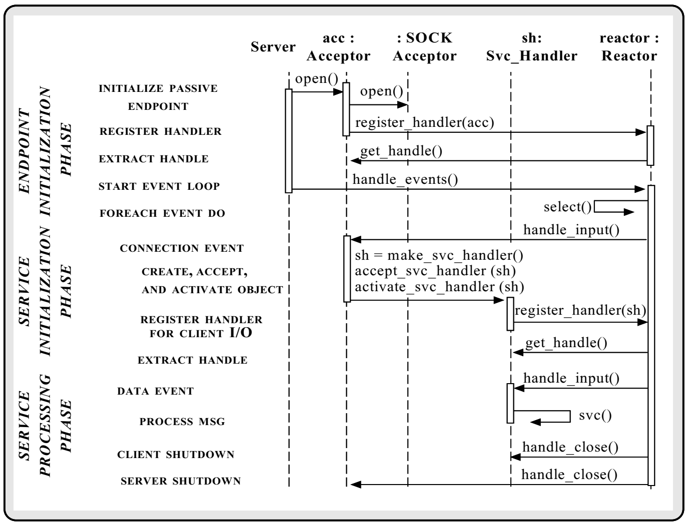

Figure 4: Collaborations Among Participants in the Acceptor Pattern

4.6 Consequences

The Acceptor pattern provides the following benefits:

- *Enhances the reusability, portability, and extensibility of connection-oriented software* – For instance, the application-independent mechanisms for passively establishing connections are decoupled from application-specific services. Thus, the application-independent mechanisms in the `Acceptor` are reusable components that know how to establish a connection passively 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 active connection establishment can be written once, placed into a class library or framework, and reused via inheritance, object composition, or template instantiation. Thus, the same passive connection establishment code need not be rewritten for each application. Services, in contrast, 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 and pernicious errors that can arise when programming with weakly typed network programming interfaces such as sockets or TLI [9].

The Acceptor pattern has the following drawbacks:

- *Additional instructions* – compared with overhead of programming to the underlying network programming interfaces directly. However, if parameterized types are used, there is no significant overhead as long as the compiler implements templates efficiently.
- *Additional complexity* – this pattern may add unnecessary complexity for simple client applications that connect with a single server and perform a single service using a single network programming interface.

4.7 Implementation

This section describes how to implement the Acceptor pattern in C++. The implementation described below is based on the ACE OO network programming toolkit [3]. In addition to illustrating how to implement the Acceptor pattern, this section shows how the pattern interacts with other common communication software patterns provided by ACE.

Figure 5 divides participants in the Acceptor pattern into the *Reactive*, *Connection*, and *Application* layers.[3] The Reactive and Connection layers perform generic, application-independent strategies for handling events and establishing connections passively, respectively. The Application layer instantiates these generic strategies by providing concrete template classes that establish connections and perform service processing. This separation of concerns increases the reusability, portability, and extensibility of this implementation of the Acceptor pattern.

There is a striking similarity between the structure of the Acceptor class diagram and the Connector class diagram shown in [2]. In particular, the Reactive layer is identical in both and the roles of the `Svc Handler` and `Concrete Svc Handler` are also very similar. Moreover, the `Acceptor` and `Concrete Acceptor` play roles equivalent to the `Connector` and `Concrete Connector` classes. In the Acceptor pattern, however, these two classes play an *passive* role in establishing a connection, rather than a *active* role.

[3] This diagram illustrates additional Booch notation: directed edges indicate inheritance relationships between classes; a dashed directed edge indicates template instantiation; and a solid circle illustrates a composition relationship between two classes.

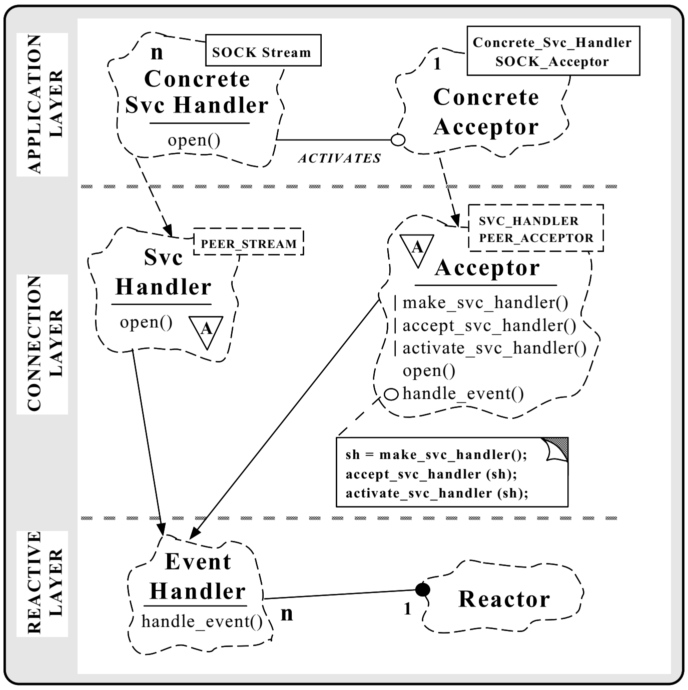

Figure 5: Layering of Participants in the Acceptor Pattern

4.7.1 Reactive Layer

The Reactive layer is responsible for handling events that occur on endpoints of communication represented by I/O handles (also known as "descriptors"). The two participants at this layer, the `Reactor` and `Event Handler`, are reused from the Reactor pattern [1]. This pattern encapsulates OS event demultiplexing system calls (such as `select`, `poll` [7], and `WaitForMultipleObjects` [11]) with an extensible and portable callback-driven object-oriented interface. The Reactor pattern enables efficient demultiplexing of multiple types of events from multiple sources within a single thread of control. An implementation of the Reactor pattern is shown in [12] and the two main roles in the Reactive layer are describe below.

- **Reactor:** This class defines an interface for registering, removing, and dispatching `Event Handler` objects (such as the `Acceptor` and `Svc_Handler`). An implementation of the `Reactor` interface provides a set of application-independent mechanisms that perform event demultiplexing and dispatching of application-specific event handlers in response to events.

- **Event Handler:** This class specifies an interface that the `Reactor` uses to dispatch callback methods defined by objects that are pre-registered to handle events. These events signify conditions such as a new connection request or the arrival of data from a connected peer.

4.7.2 Connection Layer

The Connection layer is responsible for creating a service handler, passively connecting a service handler to its peer, and activating the handler once it's connected. Since all behavior at this layer is completely generic, these classes delegate to the concrete IPC mechanism and concrete service handler instantiated by the Application layer. Likewise, the Connection layer delegates to the Reactor pattern in order to establish connections asynchronously without requiring multi-threading. The two primary roles in the Connection layer are described below.

• **Svc Handler:** This abstract class provides a generic interface for processing services. Applications must customize this class to perform a particular type of service.

```
template <class PEER_STREAM> // Concrete IPC mech.
class Svc_Handler : public Event_Handler
{
public:
  // Pure virtual method (defined by a subclass).
  virtual int open (void) = 0;

  // Conversion operator needed by
  // Acceptor and Connector.
  operator PEER_STREAM &() { return stream_; }

protected:
  PEER_STREAM stream_; // Concrete IPC mechanism.
};
```

The open method of a Svc Handler is called by the Acceptor factory after a connection is established. The behavior of this pure virtual method must be defined by a subclass, which performs service-specific initializations. A subclass of Svc Handler also determines the service's concurrency strategy. For example, a Svc Handler may employ the Reactor [1] pattern to process data from peers in a single-thread of control. Conversely, a Svc Handler might use the Active Object pattern [13] to process incoming data in a different thread of control than the one the Acceptor object used to connect it. Section 4.8 illustrates how several different concurrency strategies can be configured flexibly without affecting the structure of the Acceptor pattern.

• **Acceptor:** This abstract class implements the generic strategy for passively initializing network services. The following class interface illustrates the key methods in the Acceptor factory:

```
template <class SVC_HANDLER, // Type of service
          class PEER_ACCEPTOR> // Accepts connections
class Acceptor : public Event_Handler {
public:
  // Initialize local_addr listener endpoint
  // and register with Reactor.
  virtual int open
    (const PEER_ACCEPTOR::PEER_ADDR &local_addr,
     Reactor *reactor);

  // Factory that creates, connects, and
  // activates SVC_HANDLER's.
  virtual int handle_event (void);

  // Demultiplexing hooks used by Reactor
  virtual HANDLE get_handle (void) const;
  virtual int handle_close (void);

protected:
  // Defines the handler's creation strategy.
  virtual SVC_HANDLER *make_svc_handler (void);

  // Defines the handler's connection strategy.
  virtual int accept_svc_handler (SVC_HANDLER *);

  // Defines the handler's concurrency strategy.
  virtual int activate_svc_handler (SVC_HANDLER *);

private:
  // IPC mech. that establishes connections passively.
  PEER_ACCEPTOR peer_acceptor_;

  // Event demultiplexor.
  Reactor *reactor_;
};

// Useful "short-hand" macros used below.
#define SH SVC_HANDLER
#define PA PEER_ACCEPTOR
```

Since Acceptor inherits from Event Handler, the Reactor can automatically call back to the Acceptor's handle_event method when a connection arrives from a peer. The Acceptor is parameterized by a particular type of PEER ACCEPTOR and SVC HANDLER. The PEER ACCEPTOR provides the transport mechanism used by the Acceptor to passively establish the connection. The SVC HANDLER provides the service that processes data exchanged with its connected peer. Parameterized types are used to decouple the connection establishment strategy from the type of service handler, network programming interface, and transport layer connection initiation protocol.

The use of parameterized types helps improve portability by allowing the wholesale replacement of the mechanisms used by the Acceptor. This makes the connection establishment code portable across platforms that contain different network programming interfaces (such as sockets but not TLI, or vice versa). For example, the PEER ACCEPTOR template argument can be instantiated with either a SOCK Acceptor or a TLI Acceptor, depending on whether the platform supports sockets or TLI. An even more dynamic type of decoupling could be achieved via inheritance and polymorphism by using the Factory Method and Strategy patterns described in [4]. Parameterized types improve runtime efficiency at the expense of additional space and time overhead during program compiling and linking.

The implementation of the Acceptor's methods is presented below. To save space, most of the error handling has been omitted.

Network applications use the open method to initialize an Acceptor. This method is implemented as follows:

```
template <class SH, class PA> int
Acceptor<SH, PA>::open
  (const PEER_ACCEPTOR::PEER_ADDR &local_addr,
   Reactor *reactor)
{
  // Store pointer to a Reactor.
  this->reactor_ = reactor;

  // Forward initialization to the PEER_ACCEPTOR.
  this->peer_acceptor_.open (local_addr);
```

```
  // Register with Reactor.
  this->reactor_->register_handler (this, READ_MASK);
}
```

The `open` method is passed the `local_addr` network address used to listen for connections. It forwards this address to the passive connection acceptance mechanism defined by the `PEER ACCEPTOR`. This mechanism initializes the listener endpoint, which advertises its "service access point" (*e.g.,* IP address and port number) to clients interested in connecting with the `Acceptor`. The behavior of the listener endpoint is determined by the type of `PEER ACCEPTOR` instantiated by a user. For instance, it can be a C++ wrapper for sockets, TLI, STREAM pipes, etc.

After the listener endpoint has been initialized, the `open` method registers itself with the `Reactor`. The `Reactor` performs a "double dispatch" back to the `Acceptor`'s `get_handle` method in order to obtain the underlying `HANDLE`, as follows:

```
template <class SH, class PA> HANDLE
Acceptor<SH, PA>::get_handle
{
  return this->peer_acceptor_.get_handle ();
}
```

The `Reactor` stores this `HANDLE` internally and uses it to detect and demultiplex incoming connection from clients in order to dispatch the `Acceptor`'s `handle_event` method, which is the focal point of the `Acceptor`. As shown below, `handle_event` is a Template Method [4] that implements the strategies for creating a new `SVC HANDLER`, accepting a connection into it, and activating the service:

```
template <class SH, class PA> int
Acceptor<SH, PA>::handle_event (void)
{
  // Create a new SVC_HANDLER.
  SH *svc_handler = this->make_svc_handler ();

  // Accept connection from client.
  this->accept_svc_handler (svc_handler);

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

This method is very concise since it factors all low-level details into the parameterized types. Moreover, all of its behavior is performed by virtual functions, which allow subclasses to extend any or all of the `Acceptor`'s strategies.

The `Acceptor's` default strategy for creating `SVC HANDLERs` is defined by the `make_svc_handler` method:

```
template <class SH, class PA> SH *
Acceptor<SH, PA>::make_svc_handler (void);
{
  return new SH;
}
```

The default behavior uses a "demand strategy," which creates a new `SVC HANDLER` for every new connection. However, subclasses of `Acceptor` can override this strategy to create `SVC HANDLERs` using other strategies (such as creating an individual Singleton [4] or dynamically linking the `SVC HANDLER` from a shared library).

The `Acceptor's SVC HANDLER` connection acceptance strategy is defined by the `accept_svc_handler` method:

```
template <class SH, class PA> int
Acceptor<SH, PA>::accept_svc_handler (SH *handler);
{
  this->peer_acceptor_->accept_ (*handler);
}
```

The default behavior delegates to the `accept` method provided by the `PEER ACCEPTOR`. Subclasses can override the `accept_svc_handler` method to perform more sophisticated behavior (such as authenticating the identity of the client to determine whether to accept or reject the connection).

The `Acceptor's SVC HANDLER` concurrency strategy is defined by the `activate_svc_handler` method:

```
template <class SH, class PA> int
Acceptor<SH, PA>::activate_svc_handler (SH *handler);
{
  handler->open ();
}
```

The default behavior of this method is to activate the `SVC HANDLER` by calling its `open` method. This allows the `SVC HANDLER` to select its own concurrency strategy. For instance, if the `SVC HANDLER` inherits from `Event Handler` it can register with the `Reactor`. This allows the `Reactor` to dispatch the `SVC HANDLER's` `handle_event` method when events occur on its `PEER STREAM` endpoint of communication. Subclasses can override this strategy to do more sophisticated concurrency activations (such as making the `SVC HANDLER` an "active object" [13] that processes data using multi-threading or multiprocessing).

When an `Acceptor` terminates, either due to errors or due to the entire application shutting down, the `Reactor` calls the `Acceptor`'s `handle_close` method to enable it to release dynamically acquired resources. In this case, the `handle_close` method simply closes the `PEER ACCEPTOR`'s listener endpoint, as follows:

```
template <class SH, class PA> int
Acceptor<SH, PA>::handle_close (void)
{
  return this->peer_acceptor_.close ();
}
```

4.7.3 Application Layer

The Application layer is responsible for supplying a concrete interprocess communication (IPC) mechanism and a concrete service handler. The IPC mechanisms are encapsulated in C++ classes to simplify programming, enhance reuse, and to enable wholesale replacement of IPC mechanisms. For example, the `SOCK Acceptor`, `SOCK Connector`, and `SOCK Stream` classes used in Section 4.8 are part of the `SOCK SAP` C++ wrapper library for sockets [14]. Likewise, the corresponding `TLI_*` classes are part of the `TLI SAP` C++ wrapper library for the Transport Layer Interface [7].

SOCK SAP and TLI SAP encapsulate the stream-oriented semantics of connection-oriented protocols like TCP and SPX with a efficient, portable, and type-safe C++ wrappers.

The two main roles in the Application layer are described below.

• **Concrete Svc Handler:** This class implements the concrete application-specific service activated by a Concrete Acceptor. A Concrete Svc Handler is instantiated with a specific type of C++ IPC wrapper that exchanges data with its connected peer. The sample code examples in Section 4.8 use a SOCK Stream as the underlying data transport delivery mechanism. It is easy to vary the data transfer mechanism, however, by parameterizing the Concrete Svc Handler with a different PEER STREAM (such as a TLI Stream).

• **Concrete Connector:** This class instantiates the generic Acceptor factory with concrete parameterized type arguments for SVC HANDLER and PEER ACCEPTOR. In the sample code in Section 4.8, SOCK Acceptor is the underlying transport programming interface used to establish a connection passively. However, parameterizing the Acceptor with a different PEER ACCEPTOR (such as a TLI Acceptor) is straightforward since the IPC mechanisms are encapsulated in C++ wrapper classes. Therefore, the Acceptor's generic strategy for passively initializing services can be reused, while permitting specific details (such as the underlying network programming interface or the creation strategy) to change flexibly. In particular, note that no Acceptor components must change when the concurrency strategy is modified.

The following section illustrates sample code that instantiates a Concrete Svc Handler and Concrete Acceptor to implement the Peers described in Section 3

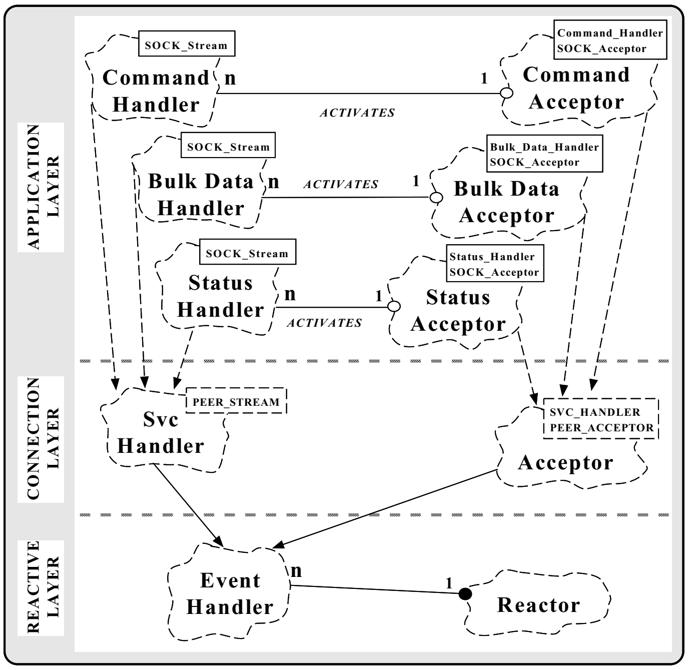

Figure 6: Structure of Participants in the Peer Acceptor Pattern

4.8 Sample Code

The sample code below illustrates how Peers described in Section 3 use the Acceptor pattern to simplify the task of passively initializing services whose connections are initiated actively by the Gateway. The Peers play the passive role in establishing connections with the Gateway (an implementation of the Gateway using the Connector pattern appears in [2]). Figure 6 illustrates how participants in the Acceptor pattern are structured in a Peer.

4.8.1 Svc Handlers for Sending and Receiving Routing Messages

The classes shown below, Status Handler, Bulk Data Handler, and Command Handler, process routing messages sent and received from a Gateway. Since these Concrete Svc Handler classes inherit from Svc Handler they are capable of being passively initialized by an Acceptor. To save space, these examples have been simplified by omitting most of the error handling code.

To illustrate the flexibility of the Acceptor pattern, each open routine in the Svc Handlers implements a different concurrency strategy. In particular, when the Status Handler is activated it runs in a separate thread; the Bulk Data Handler runs as a separate process; and the Command Handler runs in the same thread as with the Reactor that demultiplexes connection requests for the Acceptor factories. Note how changes to these concurrency strategies do not affect the architecture of the Acceptor, which is generic and thus highly flexible and reusable.

We'll start by defining a Svc_Handler that is specialized for socket-based data transfer:

```
typedef Svc_Handler <SOCK_Stream> PEER_HANDLER;
```

This class forms the basis for all the subsequent service handlers. For instance, the Status_Handler class processes status data sent to and received from a Gateway:

```
class Status_Handler : public PEER_HANDLER
{
public:
  // Performs handler activation.
  virtual int open (void) {
    // Make handler run in separate thread (note that
    // Thread::spawn requires a pointer to a static
    // method as the entry point for the thread).

    Thread::spawn (&Status_Handler::svc_run, this);
  }

  // Static entry point into the thread, which blocks
  // on the handle_event() call in its own thread.
  static void *svc_run (Status_Handler *this_) {
    // This method can block since it
```

```
    // runs in its own thread.
    while (this_->handle_event () != -1)
      continue;
  }

  // Receive and process status data from Gateway.
  virtual int handle_event (void) {
    char buf[MAX_STATUS_DATA];
    this->stream_.recv (buf, sizeof buf);
    // ...
  }

  // ...
};
```

The following class processes bulk data sent to and received from the `Gateway`.

```
class Bulk_Data_Handler : public PEER_HANDLER
{
public:
  // Performs handler activation.
  virtual int open (void) {
    // Handler runs in separate process.
    if (fork () > 0) // In parent process.
      return 0;
    else // In child process.

      // This method can block since it
      // runs in its own process.
      while (this->handle_event () != -1)
        continue;
  }

  // Receive and process bulk data from Gateway.
  virtual int handle_event (void) {
    char buf[MAX_BULK_DATA];
    this->stream_.recv (buf, sizeof buf);
    // ...
  }

  // ...
};
```

The following class processes bulk data sent to and received from a `Gateway`:

```
// Singleton Reactor object.
extern Reactor reactor;

class Command_Handler : public PEER_HANDLER
{
public:
  // Performs handler activation.
  virtual int open (void) {
    // Handler runs in same thread as main Reactor.
    reactor.register_handler (this, READ_MASK);
  }

  // Receive and process command data from Gateway.
  virtual int handle_event (void) {
    char buf[MAX_COMMAND_DATA];
    // This method cannot block since it borrows
    // the thread of control from the Reactor.
    this->stream_.recv (buf, sizeof buf);
    // ...
  }

  //...
};
```

4.8.2 Acceptors for Creating Svc Handlers

The `s_acceptor`, `bd_acceptor`, and `c_acceptor` objects shown below are `Concrete Acceptor` factories that create and activate `Status Handlers`, `Bulk Data Handlers`, and `Command Handlers`, respectively.

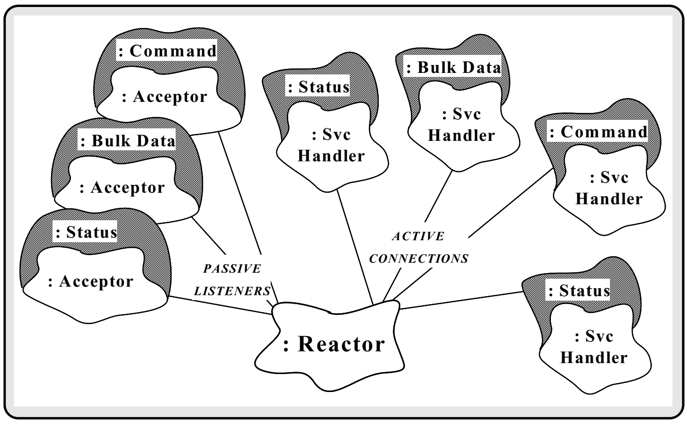

Figure 7: Object Diagram for the `Peer` Acceptor Pattern

```
// Accept connection requests from Gateway and
// activate Status_Handler.
Acceptor<Status_Handler, SOCK_Acceptor> s_acc;

// Accept connection requests from Gateway and
// activate Bulk_Data_Handler.
Acceptor<Bulk_Data_Handler, SOCK_Acceptor> bd_acc;

// Accept connection requests from Gateway and
// activate Command_Handler.
Acceptor<Command_Handler, SOCK_Acceptor> c_acc;
```

4.8.3 The main() Function

The main program initializes the concrete `Acceptor` factories by calling their `open` methods with the well-known ports for each service. As shown in Section 4.7.2, the `Acceptor::open` method registers itself with a Singleton [4] instance of the `Reactor`. The program then enters an event loop that uses the `Reactor` to detect connection requests from the `Gateway`. When connections arrive, the `Reactor` calls back to the appropriate `Acceptor`, which creates a new `PEER HANDLER` to perform the service, accepts the connection into the handler, and activates the handler.

```
// Main program for the Peer.

// Singleton Reactor object.
Reactor reactor;

int main (void)
{
  // Initialize acceptors with their well-known ports.
  s_acc.open (INET_Addr (STATUS_PORT), &reactor);
  bd_acc.open (INET_Addr (BULK_DATA_PORT), &reactor);
  c_acc.open (INET_Addr (COMMAND_PORT))& reactor);

  // Loop forever handling connection request
  // events and processing data from the Gateway.

  for (;;)
    reactor.handle_events ();
}
```

Figure 7 illustrates the relationship between Acceptor pattern objects in the `Peer` after four connections have been established. While the various `*Handlers` exchange data with the `Gateway`, the `*Acceptors` continue to listen for new connections.[4]

4.9 Known Uses

The `Reactor`, `Svc Handler`, and `Acceptor` classes described in this article are all provided as reusable components in the ACE toolkit [3]. The Acceptor pattern has been used in the following frameworks, toolkits, and systems:

- UNIX network superservers such as `inetd` [7], `listen` [8], and the `Service Configurator` daemon from the `ASX` framework [3]. These superservers 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`). The Acceptor pattern decouples the functionality in 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 monitored port, the Acceptor process accepts the request and dispatches an appropriate pre-registered handler to perform the service.
- The Ericsson EOS Call Center Management system [15] uses the Connector pattern to allow application-level Call Center Manager `Gateways` to actively establish connections with passive `Peer` hosts in a distributed system.
- The high-speed medical image transfer subsystem of project Spectrum [16] uses the Acceptor pattern to passively establish connections and initialize application services for storing large medical images. Once connections are established, applications then send and receive multi-megabyte medical images to and from these image stores.

4.10 Related Patterns

The Acceptor pattern may be viewed as a variation of the Template Method and Factory Method patterns [4]. 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 this task is decoupled from the protocol used to create the task.

The Acceptor pattern is a connection factory that uses a template method (`handle_event`) to create handlers for communication channels. The `handle_event` method implements the algorithm that passively listens for connection requests, then creates, accepts, 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.

[4]This diagram uses additional Booch notation [10], where solid clouds indicate objects and undirected edges indicate some type of link (such as a pointer or reference) exists between two objects.

5 Concluding Remarks

This article motivates the Acceptor and Connector patterns and gives a detailed example illustrating how to use the Acceptor pattern. A subsequent issue of the C++ Report [2] will illustrate how to implement the Connector pattern. UNIX versions of the Acceptor, Connector, and Reactor patterns described in this article are freely available via the World Wide Web at URL `http://www.cs.wustl.edu/~schmidt/`. This distribution contains complete source code, documentation, and example test drivers for the C++ components developed as part of the ACE object-oriented network programming toolkit [3] developed at the University of California, Irvine and Washington University. The ACE toolkit is currently being used on communication software at many companies.

Thanks to Venkata-Subbarao Kandru for comments on this paper.

References

[1] 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.

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

[3] 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.

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

[5] S. J. Leffler, M. McKusick, M. Karels, and J. Quarterman, *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison-Wesley, 1989.

[6] W. R. Stevens, *TCP/IP Illustrated, Volume 1*. Reading, Massachusetts: Addison Wesley, 1993.

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

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

[9] 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*, (Monterey, CA), USENIX, June 1995.

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

[11] H. Custer, *Inside Windows NT*. Redmond, Washington: Microsoft Press, 1993.

[12] D. C. Schmidt, "The Object-Oriented Design and Implementation of the Reactor: A C++ Wrapper for UNIX I/O Multiplexing (Part 2 of 2)," *C++ Report*, vol. 5, September 1993.

[13] 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.

[14] D. C. Schmidt, "IPC_SAP: An Object-Oriented Interface to Interprocess Communication Services," *C++ Report*, vol. 4, November/December 1992.

[15] 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.

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