Acceptor and Connector

Design Patterns for Actively and 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

An earlier version of this paper was presented at the Workshop on Pattern Languages of Object-Oriented Programs at ECOOP '95, August 7 – 1, 1995, Aarhus, Denmark.

Abstract

This paper describes a pair of patterns that enhance the reuse, extensibility, and portability of object-oriented communication software by permitting network services to evolve independently of the mechanisms used to establish connections used by the services. The Acceptor and Connector patterns decouple the passive and active establishment of a connection, respectively, from the service performed once the connection is established. In addition, the patterns decouple the service creation and concurrency strategies from the services themselves.

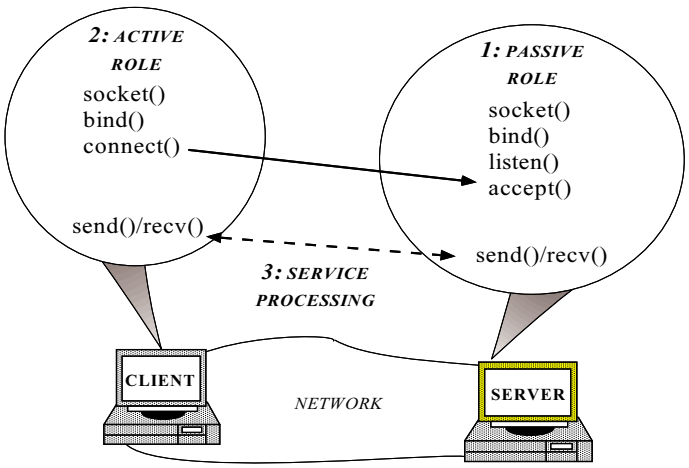


Figure 1: Active and Passive Connection roles

1 Introduction

Connection-oriented protocols (such as TCP/IP or IPX/SPX) provide reliable delivery of 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 operate when a connection is established between an active client and a passive server using the socket network programming interface and the TCP transport protocol. In this figure the server plays the passive role and the client plays the active role. 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 it is possible for the connection roles to be reversed, as discussed in Section 2.

The intent of the Acceptor and Connector patterns described in this paper is to decouple the passive and active connection roles, respectively, from the services performed once a connection is established. These patterns are motivated by the observation that the service processing performed between connected endpoints is often 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 in any manner that obeys the protocol used by the two endpoints (*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 [1] or TLI [2]) 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 processes in a distributed application.

- *The concurrency method used by the applications to execute the service* – for instance, a service can run in a single-thread, in multiple threads, or multiple processes, regardless of how the connections were established.

Although the Acceptor and Connector 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 addition forces by using asynchrony to actively establish connections with a large number of peers efficiently.

This paper is organized as follows: Section 2 motivates the Connector and Acceptor patterns by illustrating how they can be used to build connection-oriented, multi-service, application-level `Gateways`; Sections 3 and 4 describe the Acceptor and Connector patterns in detail, respectively, and Section 5 presents concluding remarks.

2 Motivation

To illustrate the Acceptor and Connector patterns, consider a connection-oriented, multi-service, application-level Gateway that is responsible for routing various types of data between `Peers` in a distributed system. `Peers` communicate with one another indirectly through the multi-service `Gateway`. The `Gateway` routes several types of data (such as status information, bulk data, and commands) that are exchanged between services running on the `Peers`. From the `Gateway`'s perspective, these services differ solely in terms of their message framing formats and payload types.

To enhance performance, a connection-oriented transport protocol (such as TCP or SPX) is used as an interprocess communication (IPC) mechanism that transmits data between the `Gateway` and the `Peers`. Each communication service sending status information, bulk data, or commands uses separate connections that are bound to unique port numbers. For example, bulk data sent from a ground station `Peer` through the `Gateway` uses a connection to a different port than status information sent by a tracking station peer is routed through the `Gateway` to a ground station `Peer`. Separating connections in this manner allows more flexible routing policies and more robust error handling if connections fail.

One way to establish connections in the connection-oriented `Gateway` is to designate the connection roles *a priori*. For example, the `Gateway` could be hard-coded to actively initiate the connections for all its services by iterating through a list of `Peers` and synchronously connecting with them. Likewise, `Peers` could be hard-coded to passively accept the connections. The active and passive connection code for the `Gateway` and `Peers`, respectively, would then be implemented with network programming interfaces (such as sockets or TLI). For example, a `Peer` would call `socket, bind, listen,` and `accept` to initialize a passive-mode listener socket. Conversely, the `Gateway`

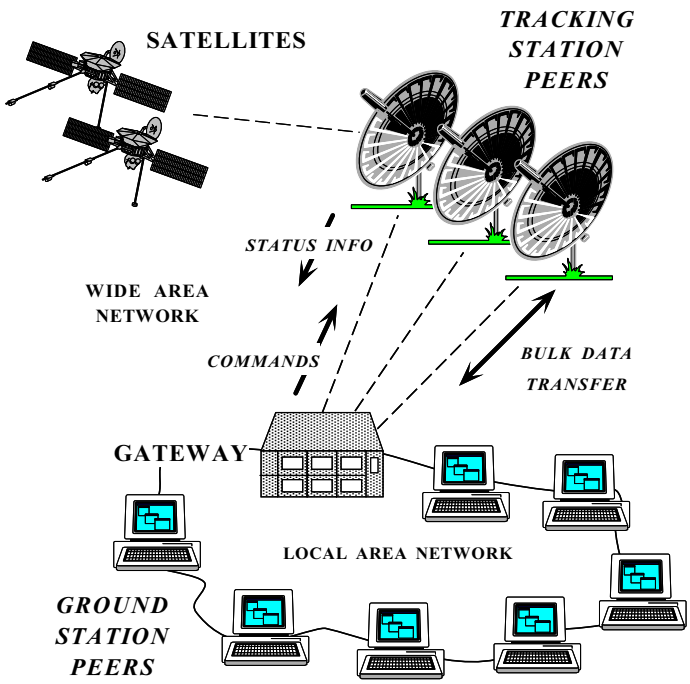


Figure 2: A Connection-oriented, Multi-service Application-level Gateway

would call `socket` and `connect` to initiate a data-mode connection socket actively. Once the connections were established the `Gateway` would route data for each type of service it provided.

However, this approach has several drawbacks:

- *Limited extensibility and reuse* – of the `Gateway` and `Peer` communication software. For example, the type of routing service (*e.g.,* status information, bulk data, or commands) performed by the `Gateway` is actually independent of the mechanisms used to establish the connection. If the connection establishment logic is too tightly coupled with the service, it is hard to reuse existing services or 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) [3] do not provide adequate type-checking since they utilize low-level I/O handles. This makes it very easy to accidentally misuse these interfaces in ways that are not 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 endsystems.

A more flexible and efficient way to design a connection-oriented, multi-service, application-level `Peers` and `Gateway` is to use the *Acceptor* and *Connector* patterns.

These patterns decouple the various types of routing services from the connection establishment mechanisms, yielding the following benefits:

- *Enhanced reuse, extensibility, and portability* – the Connector and Acceptor patterns permit key characteristics of the routing service (such as the concurrency strategy or data framing behavior) to evolve independently of the mechanisms used to establish the connections.
- *Increased type-safety* – these patterns abstract away from low-level, weakly-typed network programming interfaces. This helps prevent application developers from accidentally misusing operations incorrectly.
- *Greater Scalability* – In addition to decoupling connection establishment from service processing, the Connector pattern pattern employs asynchronous connection mechanisms that initiate and complete multiple connections concurrently. This helps to improve the efficiency of actively establishing connections simultaneously to many endpoints.

Sections 3 and 4 describe the Acceptor and Connector pattern in detail.

3 The Acceptor Pattern

3.1 Intent

Decouple the passive establishment of a connection from the service performed once the connection is established.

3.2 Also Known As

Listener

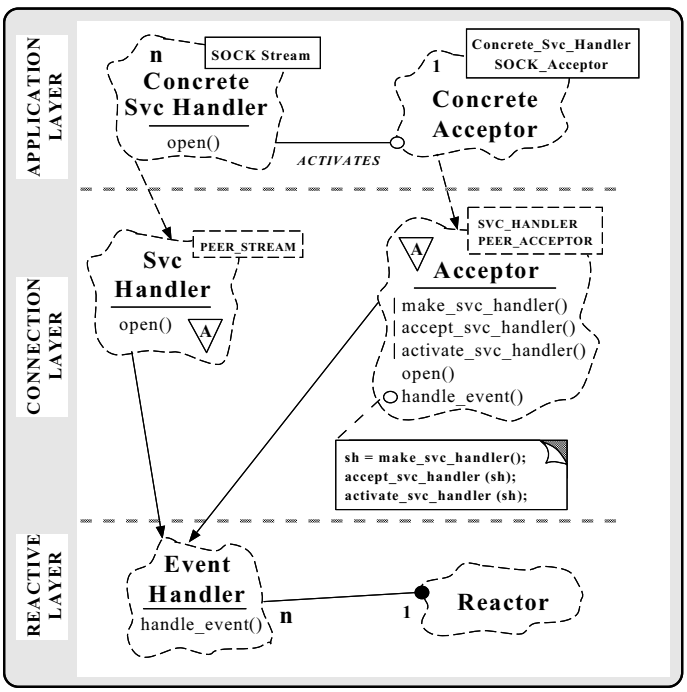
3.3 Applicability

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

- Application services (such as gateways, electronic medical imaging systems, or video-on-demand systems) are independent of the steps required to establish a connection;
- A service's concurrency strategy can be decoupled from its connection establishment mechanisms;
- Connections may arrive concurrently from different peers, but blocking or continuous polling for incoming connections on any individual peer is inefficient;

3.4 Structure and Participants

The structure of the Acceptor pattern is illustrated by the following Booch class diagram [4]. In this diagram dashed clouds indicate classes; directed edges indicate inheritance relationships between classes; an undirected edge with a hollow circle at one end indicates a uses relation between two classes and a filled circle illustrates a composition relationship between two classes; and a dashed directed edge indicates template instantiation.



The participants in the Acceptor pattern are discussed below. The `Reactor` and `Event Handler` are reused from the Reactor pattern presented in [5]. The `Reactor` defines a mechanism for registering, removing, and dispatching `Event_Handlers` (such as the `Acceptor` and `Svc_Handler` described below). The `Reactor` encapsulates OS event demultiplexing system calls (such as `select` and `poll`) with an extensible and portable callback-driven object-oriented interface. The four remaining participants in the Acceptor pattern build upon the Reactor pattern's event demultiplexing and dispatching components.

- **Reactor**
  - Defines an interface for registering, removing, and dispatching `Event Handler` objects. 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**
  - Specifies an interface used by the `Reactor` to dispatch callback methods defined by objects that are pre-registered to handle events (such as connection request events from clients).
- **Acceptor**

– This factory class assembles the resources necessary to create and activate a `Svc Handler`. An `Acceptor` implements the strategy for passively establishing connections with `Peers`. It is parameterized by concrete types that conform to the interfaces of `PEER ACCEPTOR` and `SVC HANDLER`. The `PEER ACCEPTOR` is instantiated with a transport mechanism that passively establishes connections. The `SVC HANDLER` is instantiated with a concrete type that performs the application-specific service. Parameterized types are used since the Connector pattern's connection establishment strategy is independent of the type of service and the type of connection mechanism.

- **Svc Handler**

  – Defines the interface for a service that exchanges data with a connected peer. The `Svc Handler` contains a `PEER STREAM` endpoint that is initialized by an `Acceptor` when a connection is established successfully. This endpoint is used to exchange data between a `Svc Handler` and the peer it is connected with.

- **Concrete Acceptor**

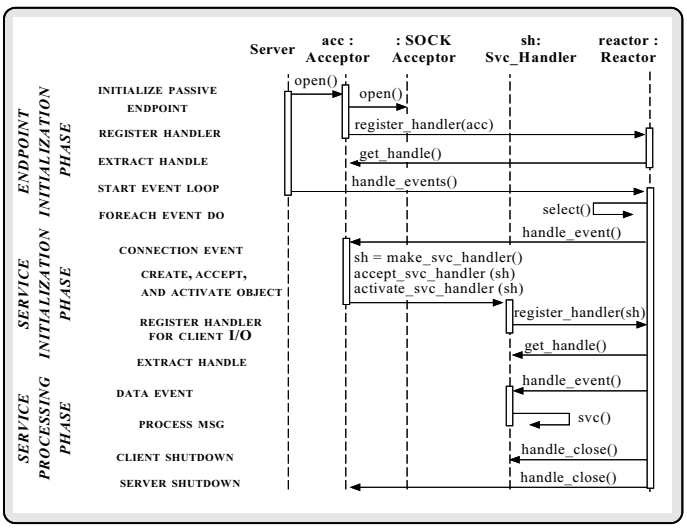
  – Instantiates the `Acceptor` factory by integrating concrete types that perform an application-specific service (*e.g.,* a `Status Handler` that processes status data) and accept connections (*e.g.,* a passive connection factory like `SOCK Acceptor` that creates `SOCK Stream` data-mode objects).[1]

- **Concrete Svc Handler**

  – Instantiates a `Svc Handler` with a concrete type (*e.g.,* `SOCK Stream`) that exchanges data with a peer. In addition, this class implements the application-specific service activated by the `Acceptor`.

3.5 Collaborations

The collaborations among participants in the Acceptor pattern are illustrated in the following object interaction diagram:



These collaborations are divided into three phases:

1. *Endpoint initialization phase* – which creates a passive-mode endpoint (encapsulated by `PEER ACCEPTOR`) that is bound to a network address (such as an IP address and port number). The passive-mode endpoint listens for connection requests from peers. This endpoint is registered with the `Reactor`, which then goes into an event loop waiting on that endpoint for connection requests to arrive from peers.
2. *Service initialization phase* – Since an `Acceptor` inherits from an `Event Handler` the `Reactor` can dispatch the `Acceptor`'s `handle_event` method when connection events arrive. When connections arrive, the `Reactor` calls back to the `Acceptor`'s `handle_event` method. This Template Method [7] performs the `Acceptor`'s `Svc Handler` activation strategy. This strategy assembles the resources necessary to create a new `Concrete Svc Handler` object, accept the connection into this object, and activate the `Svc Handler` by calling its `open` method.
3. *Event handling phase* – once activated, the `Svc Handler` processes incoming event messages arriving on the `PEER STREAM`. A `Svc Handler` will process incoming event messages using the Reactor pattern or some other form of concurrent event handling such as the Active Object pattern [8]. The concurrency strategy used by a `Svc Handler` is defined by its `open` method. Section 3.7 discusses alternative methods for implementing `Svc Handler` concurrency.

3.6 Consequences

The Acceptor pattern provides the following benefits:

- *The application-independent mechanisms for passively establishing connections are decoupled from*

[1] The `SOCK Acceptor`, `SOCK Connector`, and `SOCK Stream` classes used in this paper are part of the `SOCK SAP` C++ wrapper library for sockets [6]. `SOCK SAP` encapsulates the stream-oriented semantics of the socket network programming interface with a portable and type-safe C++ wrapper.

application-specific services – Thus, the application-independent mechanisms become reusable components that know how to demultiplex events and dispatch the `Acceptor`'s `handle_event` callback method to accept a connection passively, create a new `Svc Handler`, and activate it. In contrast, the `Concrete Svc Handler` knows how to perform an application-specific service. This separation of concerns enhances the modularity, reusability, and configurability of connection-oriented software.

For example, decoupling connection establishment from service handling allows each part to evolve independently. Typically, the strategy for establishing connections passively is 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` by a `Concrete Svc Handler`, the impact of this variation is localized.

- *Improves application portability* – Parameterizing the `Acceptor`'s mechanisms for accepting connections and performing services helps to improve portability by allowing the wholesale replacement of these mechanisms. This makes the connection establishment code portable across platforms that contain different network programming interfaces (such as sockets but not TLI, or vice versa). 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.
- *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 [3].

The Acceptor pattern has the following drawback:

- *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. Templates help to improve portability since the Acceptor pattern's connection establishment strategy is independent of the service type, network programming interface, and transport layer connection establishment protocol. An even more dynamic type of decoupling can be achieved via inheritance and polymorphism by using the Abstract Factory or Factory Method patterns described in [7]. The implementation of the Acceptor pattern described in this paper uses parameterized types to improve run-time efficiency (at the expense of additional compile-time and link-time time and space overhead).

3.7 Implementation

The Acceptor pattern may be implemented by decoupling connection establishment from service functionality as shown below. Note that both classes shown below use parameterized types to decouple connection establishment from service functionality.

3.7.1 Define a Svc Handler Interface

First, define a `Svc Handler` class that provides an interface for defining and handling services, as follows:

```
template <class PEER_STREAM, // IPC mechanism
          class PEER_ADDR> // Type of network address
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 this->stream_; }

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

`Svc Handler` inherits from the Reactor pattern's `Event Handler`. This enables its `handle_event` method to be dispatched by a `Reactor` when events occur on the `PEER STREAM`, which provides the underlying 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. This subclass is also responsible for determining the `Svc Handler`'s concurrency policy.

The Acceptor pattern does not dictate the behavior or concurrency policies of the `Svc Handler` service it creates. Delegating this decision to `Svc Handler` allows different subclasses of `Svc Handler` to select different policies without affecting existing code. For example, a `Svc Handler` service may execute in any of the following ways:

- *Run in the same thread of control* – where each `Svc Handler` object is dispatched in the same thread of control as the `Acceptor` and `Reactor` objects. Implementing this scheme is straightforward since the `Svc Handler` inherits from `Event Handler`. Therefore, the `open` method of each newly activated `Svc Handler` object can register itself with a `Reactor`.
- *Run in a separate thread of control* – In this approach, the `Reactor` serves as the master connection dispatcher within an application. When a peer connects, the `Svc Handler`'s `open` method spawns a separate thread of control. The `Svc Handler` object then processes messages exchanged over the connection within

the new slave thread. Threads are useful for cooperating services that frequently reference common memory-resident data structures shared by the threads within a process address space [9].

- *Run in a separate OS process* – This approach is similar to the multi-threading approach described above. However, the `Svc Handler` `open` method forks a separate process rather than spawn a separate thread. Communication services (such as `ftp` or `telnet`) that base their security and protection mechanisms on process ownership are typically executed in separate processes. This prevents accidental or intentional access to unauthorized resources.

3.7.2 Define an Acceptor Interface

Once the `Svc Handler` is defined, create an `Acceptor` class that implements the strategy for accepting connections and activating `Svc Handlers` to perform a specific service. The following class illustrates the key parts of an `Acceptor` factory:

```
template <class SVC_HANDLER, // Type of service
          class PEER_ACCEPTOR> // Passive Conn. Mech.
class Acceptor : public Event_Handler
{
public:
  // Bind the PEER_ACCEPTOR to the PEER_ADDR.
  virtual int open (const PEER_ACCEPTOR::PEER_ADDR &addr) {
    this->acceptor_.open (addr);
  }

  // Template Method strategy for accepting
  // and activating a service.

  virtual int handle_event (void) {
    // Create new handler for a service.
    SVC_HANDLER *sh = this->make_svc_handler ();

    // Accept connection into handler.
    this->acceptor_.accept (*sh);

    // Delegate initialization to service handler.
    sh->open (this);
  }

  // Factory method for creating a service handler.
  // Can be overridden by subclass to create service
  // handlers in various ways.

  virtual SVC_HANDLER *make_svc_handler (void) {
    return new SVC_HANDLER;
  }

protected:
  // Factory for accepting connections passively.
  PEER_ACCEPTOR acceptor_;
};
```

The `Acceptor` is parameterized by the type of service handle (`SVC HANDLER`), the type of passive connection mechanism (`PEER ACCEPTOR`), and the type of networking address (`PEER ADDR`) used by the passive connection mechanism. Its `open` method is passed the well-known network address used to listen for connections and forwards this address to the passive connection mechanism. The `handle_event` method implements the `Acceptor`'s strategy for creating a new service handler, accepting a connection into it, and activating the service. Note that this method is very concise since it factors all the low-level details into the parameterized types.

The connection establishment semantics provided by the underlying OS significantly affect the implementation of the `Acceptor`. The implementation shown above is designed to work on UNIX systems, which provide "reactive" event demultiplexing semantics [1]. The UNIX `select` and `poll` system calls indicate the subset of passive mode I/O handles that may establish connections synchronously without blocking. In contrast, Windows NT provides "proactive" event demultiplexing semantics [10]. Proactive passive connection establishment operations proceed asynchronously and do not cause the caller to block. An application may subsequently use the WIN32 `WaitForMultipleObjects` system call to determine when its outstanding asynchronous connection requests have completed. [11] provides a detailed evaluation of how differences between proactive and reactive event demultiplexing affect implementations of the Acceptor pattern on UNIX and Windows NT.

3.8 Sample Code

This section uses the `Acceptor` and `Svc Handler` classes defined in Section 3.7 to illustrate how a `Peer` passively establishes connections and activate services (Section 4.7 illustrates the components used to actively establish connections via the Connector pattern). The structure of a `Peer` designed in terms of the Acceptor pattern is shown in Figure 3.

The following three subclasses produce application-specific `Concrete Svc Handlers` for the status service, the command service, and the bulk data service utilized by `Peers`. All three classes inherit from `Svc Handler` and instantiate it with a `SOCK Stream` and `INET Addr`. 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 implements a different concurrency policy. When activated, the `Status Handler` 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 is demultiplexing connection requests for the three `Acceptor` factories.

```
// Global per-process Reactor object.
Reactor reactor;

// Processes status data from/to Gateway.

class Status_Handler
  : public Svc_Handler <SOCK_Stream, INET_Addr>
{
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).
```

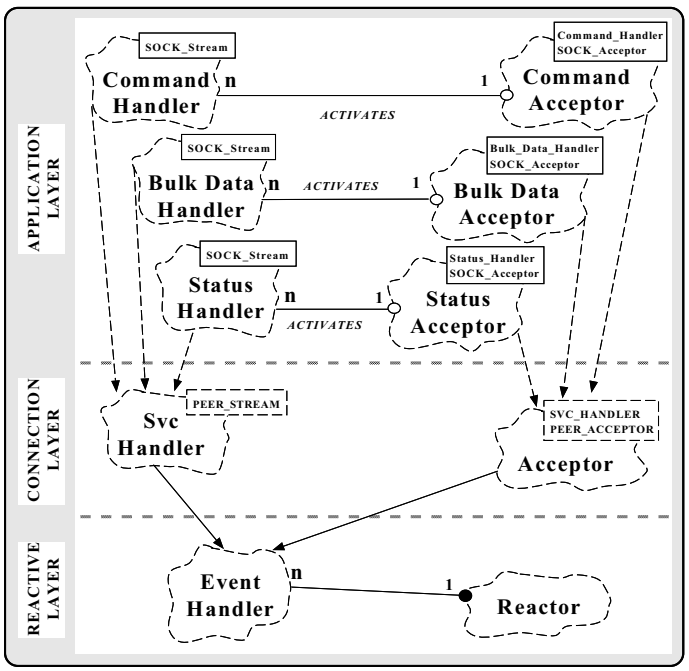


Figure 3: Structure of Participants in the Peer Acceptor Pattern

```
    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);
    // ...
  }

  // Sends status data to the Gateway.
  virtual int put (Message_Block *telem_data);

// ...
};

// Processes bulk data from/to Gateway.

class Bulk_Data_Handler
  : public Svc_Handler <SOCK_Stream, INET_Addr>
{
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);
    // ...
  }

  // Sends bulk data to the Gateway.
  virtual int put (Message_Block *bulk_data);

// ...
};

// Processes Command data from/to Gateway.

class Command_Handler
  : public Svc_Handler <SOCK_Stream, INET_Addr>
{
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);
    // ...
  }

  // Sends command data to the Gateway.
  virtual int put (Message_Block *cmd_data);
};
```

The t_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.

```
// Accept connection requests from Gateway and
// activate Status_Handler.

Acceptor<Status_Handler, SOCK_Acceptor,
         INET_Addr> t_acceptor;

// Accept connection requests from Gateway and
// activate Bulk_Data_Handler.

Acceptor<Bulk_Data_Handler, SOCK_Acceptor,
         INET_Addr> bd_acceptor;

// Accept connection requests from Gateway and
// activate Command_Handler.

Acceptor<Command_Handler, SOCK_Acceptor,
         INET_Addr> c_acceptor;
```

The main program initializes the concrete Acceptor factories with their well-known ports and registers them with a global instance of the Reactor. The program then enters an event loop that accepts connection requests from the Gateway and creates the appropriate type of Svc Handler to perform the service.

```
// Main program for the Peer.

int main (void)
{
```

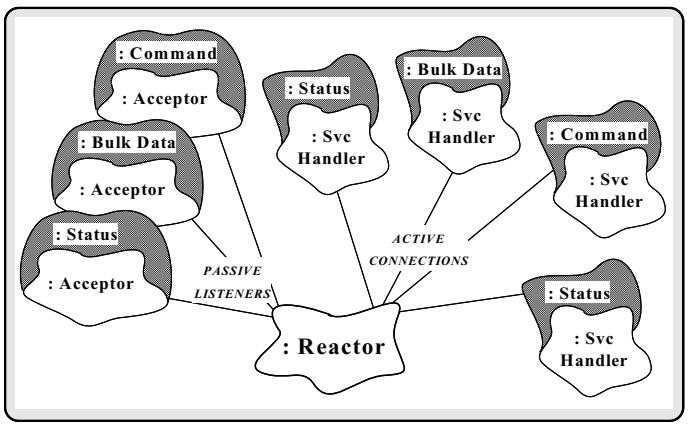


Figure 4: Object Diagram for the `Peer` Acceptor Pattern

```
  // Initialize acceptors with their well-known ports.
  t_acceptor.open (STATUS_PORT);
  bd_acceptor.open (BULK_DATA_PORT);
  c_acceptor.open (COMMAND_PORT);

  // Register Acceptors with Reactor.
  reactor.register_handler (&t_acceptor, READ_MASK);
  reactor.register_handler (&bd_acceptor, READ_MASK);
  reactor.register_handler (&c_acceptor, READ_MASK);

  // Loop forever handling events.

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

Figure 4 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. This diagram uses additional Booch notation [4], where solid clouds indicate objects and undirected edges indicate some type of link exists between two objects.

3.9 Known Uses

The `Reactor`, `Svc Handler`, and `Acceptor` classes described in this paper are all provided as reusable components in the ACE toolkit [12]. The Acceptor pattern has been used in the following frameworks, toolkits, and systems:

- UNIX network superservers such as `inetd` [1], `listen` [2], and the `Service Configurator` daemon from the `ASX` framework [12]. 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 Motorola Iridium project uses the Acceptor pattern to allow `Peers` to passively establish connections in response to active connection requests from an application-level `Gateway`.
- The Ericsson Call Center Management system [11] uses the `Acceptor` to enhance the flexibility and reuse of network management software that monitors and manages telecommunication switches across multiple hardware and software platforms.

3.10 Related Patterns

The Acceptor pattern may be viewed as a variation of the Template Method and Factory Method patterns [7]. 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 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 The Connector Pattern

4.1 Intent

Decouples active connection establishment from the service performed once a connection is established.

4.2 Applicability

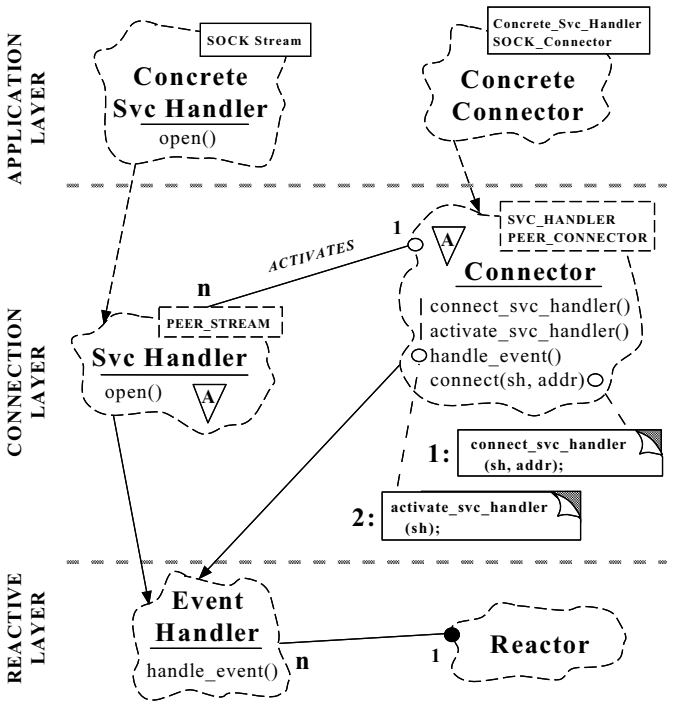
The Connector pattern is applicable in the same situations as the Acceptor pattern described in Section 3.3. In particular, use the Connector pattern when an application's communication services are independent of the steps required to establish a connection. In addition, use the Connector pattern when a connection-oriented application possesses the following characteristics:

- The use of higher-level communication frameworks (such as CORBA or DCE) is infeasible due to performance requirements or interoperability constraints and
- There are a large number of connections to establish and
- There is a long latency associated with the transmission media over which the connections are established.

By using asynchrony, the Connector pattern enables applications to actively establish connections with a large number of peers efficiently over long-delay transmission media paths (such as wide-area networks).

4.3 Structure and Participants

The structure and participants in the Connector pattern are illustrated in the following Booch class diagram. Note the structural similarity between the Connector class diagram and the Acceptor diagram shown in Section 3.4.

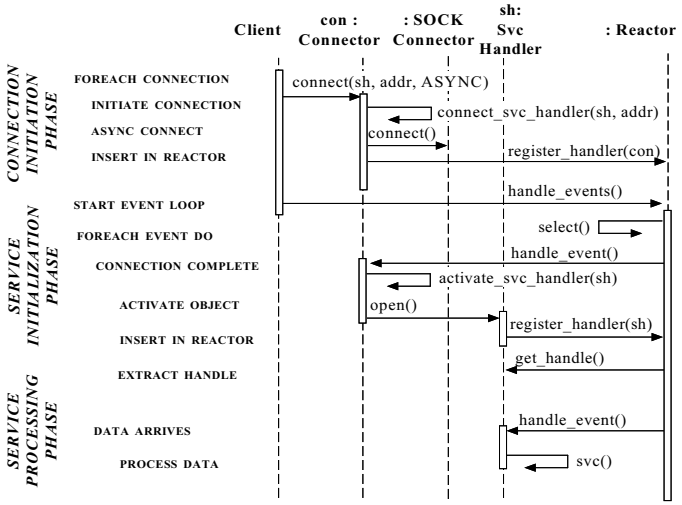


The Connector pattern reuses the `Reactor`, `Event Handler`, `Svc Handler`, and `Concrete Svc Handler` described in Section 3.4. In addition, the `Connector` and `Concrete Connector` described below are similar to the `Acceptor` and `Concrete Acceptor`. In the Connector pattern, however, these two components play an *active* role in establishing a connection, rather than a passive role.

- **Connector**
  - This factory class assembles the resources necessary to create and activate a `Svc Handler`. The `Connector` implements the strategy for actively establishing connections with peers asynchronously. It is parameterized by a particular type of `SVC HANDLER` (which performs a service in conjunction with a connected peer) and a `PEER CONNECTOR` (which is the underlying IPC transport mechanism used to actively establish the connection synchronously or asynchronously). Parameterized types are used since the Connector pattern's connection establishment strategy is independent of the type of service and the type of connection mechanism.
- **Concrete Connector**
  - A programmer supplies the `SVC HANDLER` and `PEER CONNECTOR` type arguments to produce a `Concrete Connector`. In this example `SOCK Connector` is shown as the underlying transport programming interface for establishing a connection. By parameterizing the `Connector` with a `PEER CONNECTOR` (such as `SOCK Connector` or `TLI Connector`) it is straightforward to reuse the `Connector`'s actively connection establishment strategy, while selectively varying the transport mechanism.

4.4 Collaborations

The collaborations among participants in the Connector pattern are illustrated are the following object interaction diagram:



These collaborations are divided into three phases:

1. *Connection initiation phase* – which is used to actively connect a set of `Svc Handlers`. If connections are established using non-blocking-mode they are initiated asynchronously, otherwise they are initiated synchronously.
2. *Service initialization phase* – which activates `Svc Handlers` when connections complete successfully. The `Connector` maintains a table of `Svc Handlers` whose connections are pending completion. When a connection completes, the `Connector` removes the connected `Svc Handler` from its table and activates it.

3. *Event handing phase* – depending on the `open` method of `Svc Handler`, this phase may employ the Reactor pattern or Active Object pattern to process incoming events. For example, when commands arrive at a `Command Handler` in the `Gateway`, the `Reactor` dispatches `Event Handlers` to frame the commands, determine outgoing routes, and deliver the commands to their destinations.

4.5 Consequences

The Connector pattern has the same consequences as the Acceptor pattern described in Section 3.6. In addition, it provides the following benefits:

- *Initiates a set of connections asynchronously to effectively utilize the inherent parallelism in the network and hosts* – For instance, a large distributed system may have several hundred `Peers` connected to a single `Gateway`. One method of connecting all these `Peers` to the `Gateway` is to use synchronous connection mechanisms. However, the round trip delay for a 3-way TCP connection handshake over a long-delay wide-area network (WAN), such as a geosynchronous satellite, may take several seconds per handshake. Therefore, if synchronous mechanisms are used to establish connections sequentially, unnecessary delays occur since the inherent parallelism of the network and computers is underutilized. By using asynchrony, the Connector pattern allows an application to actively establish connections with a large number of peers efficiently over long-delay transmission media paths.
- *Interoperates with existing transport layer protocols and network programming interfaces* – Using higher-level distributed object computing (DOC) frameworks like CORBA or OODCE can eliminates the need for the Connector and Acceptor patterns. DOC frameworks perform these operations implicitly while binding to and activating a service implementation. However, DOC frameworks have two drawbacks:
  1. *Interoperability* – DOC frameworks generally do not provide interfaces that allow applications to interoperate directly with transport protocols (such as TCP or SPX) using network programming interfaces (such as sockets or TLI). In contrast, the Connector and Acceptor patterns interoperate smoothly with existing transport layer protocols and network programming interfaces.
  2. *Scalability* – DOC frameworks typically do not provide mechanisms for efficiently establishing connections in parallel with a large number of peers.

4.6 Implementation

This section describes the steps used to implement the Connector pattern.

4.6.1 Define an Connector Interface

This pattern is implemented similarly to the Acceptor pattern described in Section 3.7. In particular, the `Svc Handler` class is exactly the same. Therefore, start by defining a class that implements the `Connector` strategy, as follows:

```
template <class SVC_HANDLER, // Type of service
          class PEER_CONNECTOR> // Active Conn. Mech.
class Connector : public Event_Handler
{
public:
  // Bind the PEER_CONNECTOR to the PEER_ADDR.
  virtual int open (const PEER_CONNECTOR::PEER_ADDR &addr) {
    this->connector_.open (addr);
  }

  // Template Method for actively connecting
  // and activating a service.

  virtual int connect (SVC_HANDLER *sh,
                       const PEER_ADDR &addr,
                       int blocking = 0) {
    if (this->connector_.connect (*sh, addr,
                                  blocking) == -1) {
      if (blocking == ACE_NONBLOCK
          && errno == EWOULDBLOCK)
        // If the connection hasn't completed and
        // we are using non-blocking semantics then
        // register ourselves with the Reactor so
        // that it will callback when the
        // connection is complete.
        this->register_handler (sh);
    }
    // Activate if we connect immediately.
    else
      sh->open ();
  }

  // Register the SVC_HANDLER with the map of
  // pending connections so that it can be activated
  // when the connection completes.

  virtual int register_handler (SVC_HANDLER *sh) {
    reactor.register_handler (this, WRITE_MASK);
    this->handler_map_.bind (this->get_handle (), sh);
  }

  // Template Method strategy for activating a
  // SVC_HANDLER whose non-blocking connection
  // has completed.

  Connector::handle_event (HANDLE sd) {
    SVC_HANDLE *sh = 0;

    // Locate the SVC_HANDLER corresponding
    // to the HANDLE.
    this->handler_map_.find (sd, sh);

    // Transfer I/O handle to SVC_HANDLE *.
    sh->set_handle (sd);

    // Remove sd from Reactor.
    this->reactor_->remove_handler (sd, WRITE_MASK);

    // Remove sd from the map.
    this->handler_map_.unbind (sd);

    // We are finally connected, so activate handler.
    sh->open ();
  }

protected:
  // Factory that establishes connections actively.
  PEER_CONNECTOR connector_;

  // Map_Manager is a collection class that maps
  // external ids (like HANDLEs) into internal
```

```
  // ids (like SVC_HANDLER *s).
  Map_Manager<HANDLE, SVC_HANDLER *> handler_map_;
};
```

This class is parameterized by the type of service handler (`SVC HANDLER`), the type of active connection mechanism (`PEER CONNECTOR`), and the type of networking address (`PEER ADDR`) used by the active connection factory. The `open` method simply forwards the well-known network address to the active connection factory.

To efficiently connection to many `Peers` simultaneously, the Connector pattern must be able to actively establish connections without blocking. One way to implement this is to use asynchronous I/O mechanisms provided by the operating system and protocol stack. Typically, the concrete `PEER CONNECTOR` object provides the low-level mechanisms for initiating connections asynchronously (*e.g.,* using sockets set into non-blocking mode). The `Connector` class provides a context for storing `Svc Handlers` whose connections are pending. Since `Svc Handler` inherits from `Event Handler`, the `Reactor` can be used to call back to the `handle_event` method when a connection completes either successfully or unsuccessfully. An asynchronous establishment may fail to connect if the remote host is down or is refusing service.

Most operating systems will try to establish a connection to a non-existent network endpoint for a fairly long period of time before reporting failure. To limit the amount of time spent waiting to connect, an implementation of the Connector pattern may allow an initiator to specify a time limit. Since a `Reactor` can demultiplex time-based events as well as I/O-based events, it can be used to implement timeouts.

4.7 Sample Code

This sample code illustrates how the Connector pattern is used to implement the connection establishment strategy of the `Gateway` described in Section 2. The `Gateway` plays the active role in establishing connections with `Peers` (an implementation of a `Peer` using the Acceptor pattern appears in Section 3.7). Figure 5 illustrates how participants in the Connector pattern are structured in the `Gateway`. The main classes in the `Gateway` are described below.

The `Connector` class is the reusable component presented in Section 4.6. The `Svc Handler`, `Template Handler`, `Bulk Data Handler`, and `Command Handler` are similar to the classes illustrated in Section 3.8. The primary difference is that the `Gateway` handlers are responsible for routing to one or more destination endpoints. In contrast, the `Peer` handlers perform the processing associated with a particular endpoint. In general, however, the core interfaces and behaviors of the `Peer` and `Gateway Svc Handlers` are very similar. This illustrates how the use of the Connector and Acceptor patterns enables services to be reused independently of whether connections are established actively or passively.

The program shown below illustrates how the `Gateway` uses the Connector pattern to simplify the task of establishing

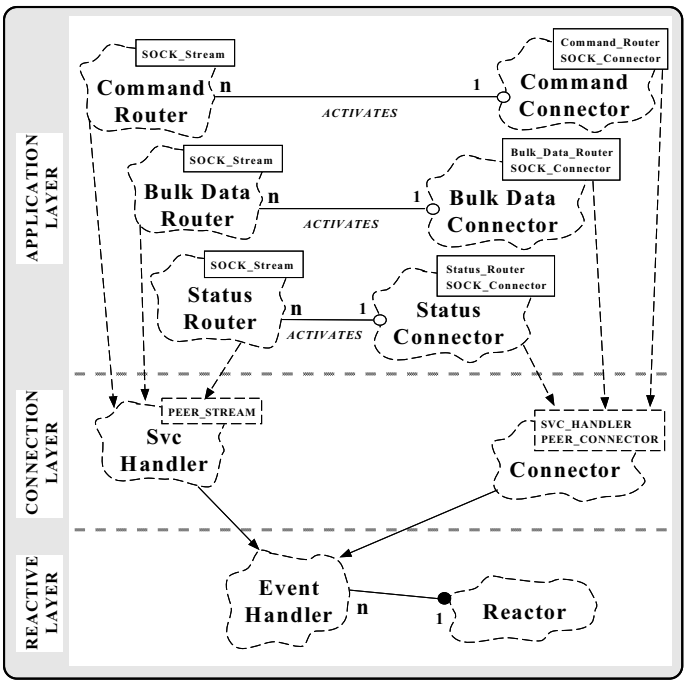


Figure 5: Structure of Participants in the `Gateway` Connector Pattern

connections with a large number of `Peers`. All connections are initiated using the following `Status`, `Bulk Data`, and `Command Connector` factories:

```
// Status connection factory.
Connector<Status_Handler, SOCK_Connector,
          INET_Addr> t_connector;

// Bulk data connection factory.
Connector<Bulk_Data_Handler, SOCK_Connector,
          INET_Addr> bd_connector;

// Command connection factory.
Connector<Command_Handler, SOCK_Connector,
          INET_Addr> c_connector;
```

The main program for the `Gateway` is shown below. It uses the `initialize_handlers` function to read a list of `Peer` addresses (consisting of IP addresses and port numbers) from a configuration file. This function creates the `Status`, `Bulk Data`, and `Command Handlers` that route messages through the `Gateway`. Once the `Handlers` are initialized, the `Connector` factories defined above initiate all the connections asynchronously.

```
// Main program for the Gateway.

int main (void)
{
  Status_Handler *t_handlers;
  Bulk_Data_Handler *bd_handlers;
  Command_Handler *c_handlers;

  // ... Acquire lists of Status_Handlers,
  // Bulk_Data_Handlers, and Command_Handlers
  // from a config file....

  initialize_handlers (t_handlers, bd_handlers,
                       c_handlers);
```

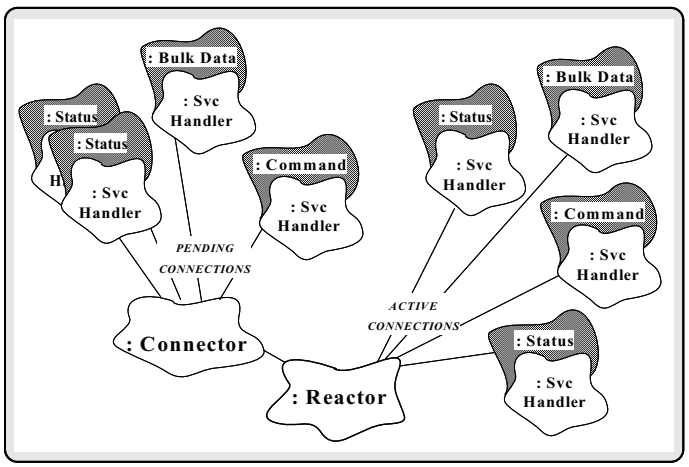


Figure 6: Object Diagram for the `Peer` Connector Pattern

```
  // Iterate through all the handlers and
  // initiate connections using non-blocking mode.

  for (int i = 0; t_handlers[i] != 0; i++)
    t_connector.connect (*t_handlers[i],
                         t_handlers[i]->addr (),
                         ACE_NONBLOCK);

  for (int i = 0; bd_handlers[i] != 0; i++)
    bd_connector.connect (*bd_handlers[i],
                          bd_handlers[i]->addr (),
                          ACE_NONBLOCK);

  for (int i = 0; c_handlers[i] != 0; i++)
    c_connector.connect (*c_handlers[i],
                         c_handlers[i]->addr (),
                         ACE_NONBLOCK);

  // Loop forever handling events.

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

Connections complete concurrently via `handle_event` callbacks within the `Reactor`'s event loop. Figure 6 illustrates the relationship between objects in the `Peer` after four connections have been established. Four other connections that have not yet completed are owned by the `Connector`. When all connections are completely established, the `Gateway` begins to route messages sent to it by `Peers`.

4.8 Known Uses

The Connector pattern has been used in the following toolkits and systems:

- The Motorola Iridium system uses the Connector pattern to allow an application-level `Gateway` to actively establish connections with passive `Peer` hosts in a distributed system.
- The high-speed medical image transfer subsystem of project Spectrum [13] uses the Connector pattern to establish application connections with image stores. Once connections are established, applications then send and receive multi-megabyte medical images to and from these image stores.

4.9 Related Patterns

Like the Acceptor pattern, the Connector pattern may be viewed as a variation of the Template Method and Factory Method patterns [7]. The Connector pattern is a Factory that uses Template Methods to create handlers for communication channels. The `connect` method implements a strategy for actively connecting to peers and activating a service handler when the connection is established. The connected service handler then performs its tasks using data exchanged on the connection. Thus, the service is decoupled from the protocol used to establish the connection.

5 Concluding Remarks

This paper describes the Acceptor and Connector patterns and gives an example illustrating how the two patterns work together. These two patterns are duals of each other, in that the Connector handles the "active" side of connection establishment and the Acceptor handles the "passive" side. Thus, the intent, applicability, structure, collaborations, and consequences are very similar.

Based on experience with these patterns on the Motorola, Ericsson, and Kodak projects, developers seem to grasp the Acceptor pattern more quickly than the Connector pattern. This may stem from the fact that clients in many connection-oriented applications interact with a relatively small number of servers. Therefore, they do not require the additional flexibility offered by the Connector pattern. In addition, the asynchronous form of the Connector pattern is more difficult to understand due to the fact that many developers have less familiarity with asynchronous interfaces and behavior.

UNIX versions of the Acceptor, Connector, and Reactor patterns 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 [12] developed at the University of California, Irvine and Washington University. The ACE toolkit is currently being used on communication software at many companies including Bellcore, Siemens, DEC, Credit Suisse, Motorola, Ericsson, and Kodak Health Imaging Systems.

References

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

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

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

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

[5] 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.

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

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1994.

[8] 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.

[9] J. Eykholt, S. Kleiman, S. Barton, R. Faulkner, A. Shivalingiah, M. Smith, D. Stein, J. Voll, M. Weeks, and D. Williams, "Beyond Multiprocessing... Multithreading the SunOS Kernel," in *Proceedings of the Summer USENIX Conference*, (San Antonio, Texas), June 1992.

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

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

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

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